Diss. ETH No. 15608

A Flexible System for Visual Service Composition

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
CESARE PAUTASSO
Laurea in Ingegneria Informatica - Politecnico di Milano
born December 9, 1975
citizen of Italy

accepted on the recommendation of
Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. Stefano Ceri, co-examiner

2004

Acknowledgments

This dissertation would have never been finished without the help of many people, to whom I would like to express my sincere gratitude.

First of all, Prof. Gustavo Alonso. He trusted my ability to complete this work even before I had started it. I also thank him for his good advice keeping me focused on such an interesting research topic. I was privileged to work in his international research group, indeed one of the best possible working environments.

Many thanks also go to Prof. Stefano Ceri for accepting to be my co-referent and for his flexibility in setting up the defense in "teledidattica". I was also truly honored for his invitation to go back to the Politecnico di Milano to present my work and receive plenty of valuable feedback.

During the past four years I was lucky to collaborate with my friend Win Bausch, who has a rare gift: everytime we would meet to discuss something we came away from the sparkling conversation with new ideas. This dissertation is the result of our teamwork.

I would also like to thank Amaia Lazcano for her feedback on the early developments of the visual language; Andrei Popovici for his suggestion about visual comments; Biörn Biöernstad for his insight about messaging systems; Andreas Frei for confirming the difference between time and space; Patrick Stuedi for testing JOpera with his large scale simulations.

Credit should also be given to all of the students who completed their Semester- and Diplomarbeits within the BioOpera and JOpera projects: Martin Grueter, Pedro Pablo Gomez Portilla, Andi Hao Zhou, Reto Schaeppi, Christian Rupp, Markus Haller, Antonio Caliano, Axel Wathne, and Jared Schirm. Particularly, Andreas Bur, Nicholas Born, Philip Frey, and Patrick Moor were brave enough to attempt to port the system to Eclipse in one semester.

I would also like to acknowledge an old school friend of mine for his honesty: he was one of the very few people telling me upfront that he would not have enough time to read the draft. Spero adesso la leggerai, Manu!

I dedicate this dissertation to my family. This is a very small achievement compared with all of their love and support during my entire life as a student. A special thank you goes to my brother, Marco, a most helpful bibliographer.

Finally, I am deeply thankful for the patience, trust and encouragement of the Esperanza of my future. Te agradezco de veras, también para llegar un paso más cerca cada año. If I could finish this, you will too!

Zurich, July 2004

Contents

Acknowledgments i

Abstract xi

Estratto xiii

1. Introduction 1
- 1.1. Motivation . . . 1
- 1.2. Contributions . . . 2
- 1.3. Structure . . . 3

2. Related Work 5
- 2.1. Visual Programming Languages . . . 5
- 2.2. Software Composition . . . 7
  - 2.2.1. Web service composition . . . 8
- 2.3. Process Modeling Languages . . . 10
  - 2.3.1. Business Process Modeling and Execution Language for Web Services . . . 13
- 2.4. Process Management Systems . . . 15
  - 2.4.1. About the JOpera project . . . 16

I. The JOpera Visual Composition Language 19

3. JOpera Visual Composition Language 21
- 3.1. Motivation . . . 21
- 3.2. Processes and Tasks . . . 23
- 3.3. Data Flow . . . 23
  - 3.3.1. Bindings . . . 24
- 3.4. Control Flow . . . 25
  - 3.4.1. Conditions . . . 25
  - 3.4.2. Synchronization . . . 26
  - 3.4.3. Exception Handling . . . 26
- 3.5. Iteration . . . 31
  - 3.5.1. List-based Loops . . . 31
  - 3.5.2. Control Flow Loops . . . 32

3.5.3. Recursion . . . 33
3.6. Comments . . . 33
3.7. Reflection . . . 34
3.7.1. System Parameters . . . 35
3.7.2. System Services . . . 36
3.8. Discussion . . . 43

4. Component Types **45**
4.1. Motivation . . . 45
4.2. Component Meta-Model . . . 47
4.2.1. Data Flow Mapping . . . 48
4.2.2. Abstract Service Types . . . 49
4.3. Web Services . . . 50
4.3.1. SOAP . . . 50
4.3.2. HTTP . . . 54
4.4. Shell Commands . . . 58
4.5. Java . . . 59
4.5.1. Java Scripts . . . 60
4.5.2. Local Java Method Calls . . . 60
4.5.3. Remote Method Invocations . . . 61
4.5.4. External Java Programs . . . 61
4.6. Script Components . . . 62
4.6.1. Scripts . . . 63
4.6.2. SQL . . . 63
4.7. XML Data Manipulation . . . 65
4.7.1. XML Components . . . 66
4.7.2. XPath Queries . . . 67
4.7.3. Style Sheet Transformations . . . 67
4.8. System Components . . . 73
4.8.1. Echo . . . 73
4.8.2. Process Invocation . . . 74
4.9. Cluster Computing . . . 75
4.9.1. PBS . . . 75
4.9.2. BioOpera . . . 76
4.10. Messaging Components . . . 79
4.10.1. Email . . . 80
4.10.2. JMS . . . 80
4.11. BPEL Basic Activities . . . 83
4.12. Workflow Tasks . . . 84
4.13. Discussion . . . 85

5. Opera Modeling Language **87**
5.1. Meta-Meta Model . . . 87
5.2. Structure of the Opera Modeling Language . . . 91
5.3. Elements of the Opera Modeling Language . . . 94

5.3.1. Root Element . . . 94
5.3.2. Abstract Elements . . . 95
5.3.3. Process Elements . . . 97
5.3.4. Data Flow Elements . . . 101
5.3.5. JOpera Visual Composition Language (JVCL) Elements . . . 105
5.3.6. Program Library Elements . . . 112
5.3.7. Component Type Modeling Elements . . . 113
5.4. Discussion . . . 114

II. The JOpera System 117

6. Compiler 119
6.1. Motivation . . . 119
6.2. Compiler's Architecture . . . 121
6.3. Mapping to OCR . . . 123
6.3.1. Data flow mapping . . . 123
6.4. BPEL Mapping . . . 125
6.4.1. Mapping to BPEL . . . 126
6.4.2. Mapping from BPEL . . . 128
6.5. Mapping to Java . . . 129
6.5.1. Process Navigation . . . 129
6.5.2. Process Instantiation . . . 130
6.5.3. Task State Diagram . . . 132
6.5.4. Process Template Plugin Interface . . . 137
6.6. Discussion . . . 143

7. Architecture 147
7.1. Motivation . . . 147
7.2. Visual Development Environment . . . 148
7.2.1. Development cycle . . . 149
7.2.2. Visual scalability . . . 150
7.2.3. Architecture . . . 150
7.3. Process Execution Kernel . . . 154
7.3.1. Architecture . . . 154
7.3.2. Threading issues . . . 157
7.3.3. Deployment Scenarios . . . 160
7.3.4. Parallel Navigation . . . 162
7.4. Supporting Heterogeneous Services . . . 163
7.4.1. Describing a task execution request . . . 165
7.4.2. Dispatching a task execution request . . . 167
7.4.3. Service invocation patterns . . . 168
7.5. API . . . 172
7.5.1. Process Control . . . 173
7.5.2. Program Library Management . . . 175

7.6. Discussion 176

8. Measurements **179**
8.1. Service Invocation Overheads 179
8.1.1. Results 180
8.1.2. Discussion 181
8.2. Visual Adaptation of Mismatching Interfaces 183
8.2.1. Results 183
8.2.2. Discussion 184
8.3. Scalability and Reliability 184
8.3.1. Results 186
8.4. Discussion 193

9. Conclusion **195**
9.1. Summary 195
9.2. Outlook 199

A. Opera Modeling Language Schema **201**

B. JOpera Compiler Output **209**

Bibliography **213**

Index **227**

Curriculum Vitae **229**

List of Examples

3.1. Book Prices 27
3.2. Late Binding 36
3.3. Cluster Resource Reservation 37
3.4. Reliable Service Invocation 39
4.1. Stock Quote Currency Conversion 55
4.2. Google Search 67
4.3. Mismatching Services Adaptation 69
4.4. Parallel Image Rendering 77
4.5. Asynchronous Process Call 81
6.1. Compiling the Stock Quote Currency Conversion Process 138

List of Figures

2.1. Summary of the process modeling languages presented in this dissertation 11
2.2. Evolution of the BPEL4WS specification 14

2.3. Evolution of the OPERA system. 16

3.1. Syntax definition for the Activity and the SubProcess 23
3.2. Data flow graph syntax 24
3.3. Control flow graph syntax 25
3.4. Control flow with exception handler 27
3.5. Control flow graph of the BookPrices process 28
3.6. First data flow view of the BookPrices process 29
3.7. Second data flow view of the BookPrices process 30
3.8. Main data flow view of the `QueryBookPrice` process 30
3.9. Data flow syntax of the list-based loops 31
3.10. Data flow view of a process to compute the factorial of an integer value 33
3.11. Example of system parameters and properties 34
3.12. Data flow view of the late binding example 37
3.13. Data flow view of the cluster resource reservation example 38
3.14. Sequential invocation of alternative services 40
3.15. Parallel invocation of alternative services 41
3.16. Split/Merge Options 42

4.1. Data flow interface of a component 48
4.2. Data flow mapping inside a component 48
4.3. Summary of the System Parameters of some the component types modeled in this chapter 52
4.4. Data flow view of the `ConvertQuote` process 55
4.5. Data flow view of the `ConvertAmount` process 56
4.6. Data flow view of the `StockQuoteConvert` process 57
4.7. Example of XML Processing with the JVCL and X-Path. 68
4.8. Example of XML Processing with the JVCL only 69
4.9. XML Schema definition for the `Adresse` type. 70
4.10. XML Schema definition for the `Address` type. 70
4.11. Visual mapping between two XML complex types 71
4.12. Equivalent XSL mapping 72
4.13. Control flow view of the `RenderImage` process 77
4.14. Data flow view of the first part of the `RenderImage` process 77
4.15. Data flow view of the second part of the `RenderImage` process 78
4.16. Data flow view of the `ClientProcess` 82
4.17. Data flow view of the `ServerProcess` 82

5.1. Mapping UML to an XML document 88
5.2. Mapping UML to an XML schema 89
5.3. Summary of the aggregation relationships between OML elements . . 90
5.4. Summary of the inheritance relationships between OML elements . . 91
5.5. Summary of the reference relationships between OML elements 92
5.6. Complete UML class diagram of the OML elements 93
5.7. Basic structure of an OML document 94

5.8. Abstract Elements: Object and Named Object 95
5.9. Structure of Processes and Tasks. 101
5.10. Structure of the data flow graph of a process. 102
5.11. Structure of the elements of the JOpera Visual Composition Language. 105
5.12. Relationship between the Opera Modeling Language and the JOpera Visual Composition Language. 108
5.13. Example on how a JVCL control flow view is stored in the underlying OML model 109
5.14. Structure of the Program and Component type library. 112
5.15. Simplified Task Model 114

6.1. Alternative approaches to process execution. 120
6.2. Multi-stage architecture of the OML compiler 122
6.3. JVCL to OCR: data flow 124
6.4. JVCL to BPEL: service invocation 126
6.5. JVCL to BPEL: control flow 127
6.6. Process navigation actions 130
6.7. Simplified state diagram of a task instance 133
6.8. Full state diagram of a task instance 135
6.9. Process Template Plugin Interface 137

7.1. Overview of the JOpera Visual Development Environment and the JOpera Kernel 148
7.2. Architecture of the JOpera Visual Development Environment 151
7.3. Architecture of a Monolithic Process Execution Kernel 154
7.4. State Information Storage Implementations for the monolithic kernel 156
7.5. Main Process Execution Loop 158
7.6. Architecture of a Distributed Kernel 160
7.7. The dispatcher as a container of task execution subsystem plugins . . 163
7.8. Interface definition of the task execution subsystem 164
7.9. Immediate service invocation 168
7.10. Synchronous service invocation 169
7.11. Asynchronous service invocation 170
7.12. Scheduled synchronous service invocation 171
7.13. Scheduled asynchronous service invocation 171
7.14. Architecture of a Peer to Peer Process Execution Kernel 177

8.1. Service Invocation Overhead for different component types 181
8.2. Performance of a visual mapping 183
8.3. Performance degradation of a centralized process support system under increasingly large workloads 186
8.4. Throughput degradation of a centralized process support system under increasingly large workloads 187
8.5. Scalability of the process instantiation 188
8.6. Memory requirements for process instantiation 189

8.7. Scalable navigation: throughput 191
8.8. Scalable navigation: event queue 192
8.9. Scalable navigation: batch execution time 193

List of Tables

2.1. Workflow patterns supported by the Opera process Modeling Language 12

4.1. Component Types Summary 51

6.1. Task State Transitions 134
6.2. Process State Definition Rules 136

7.1. Deployment Scenarios 162
7.2. Design options to describe a task execution request. 166
7.3. Summary of the JOpera Process Control API 173
7.4. Summary of the JOpera Program Library Management API 176

8.1. Service Invocation Mechanisms to be compared 180
8.2. Workload Control Variables 185

Abstract

This dissertation brings together ideas of different research areas. First of all, we propose the application of visual languages to service composition. In order to connect basic services of various kinds into a larger system, their interactions along the time dimension are defined with the JOpera Visual Composition Language. As opposed to the textual or XML-based syntax of existing approaches, our language features a very simple graphical notation. This visual syntax is used to specify the data flow and control flow graphs linking the various service invocations. This way, it becomes possible to rapidly build distributed applications out of a set of reusable services by literally drawing the interactions between them. To achieve this, we present how usability features such as automatic, incremental graph layout and visual scalability features such as multiple views have been driving the design of JOpera's visual service composition environment. To provide support for realistic application scenarios, we have also included recursion, iteration and reflection constructs with minimal changes to the syntax of the visual language. Supported by the JOpera system, our visual language for service composition has been applied to many scenarios, as documented by the examples shown throughout the dissertation.

Underneath the visual syntax, our approach to modeling service composition is based on the concept of process. In this dissertation we borrow the notion of business process so that it can be extended to model service oriented architectures. Thus, the structure of a process defines the partial order of invocation of its services, the data exchanges between them and the necessary failure handling behavior. In this context, an important contribution of this dissertation is the idea that a composition language should be orthogonal with respect to the types of components that are employed. More precisely, in our approach, composition is defined at the level of service interfaces. Therefore, a process is completely independent from the mechanisms and protocols used to access the implementation of its services. In other words, we introduce a composition language which is not limited to describing how components of a specific type (e.g., Web services) should be composed. Instead, in our open component meta-model, we generalize the notion of service by abstracting common features among a large set of different component types. This abstraction has several important implications. By supporting a large and open set of types of services, the composition language is simplified because many constructs (e.g., modeling synchronous or asynchronous service invocation) can be shifted from the composition language to the component meta-model. Also, the service composer is free to choose the most appropriate mechanism to access the functionality of an existing service. Thus, the runtime overhead of a service invocation can be minimized

as it becomes possible to choose the most efficient access mechanism.

This optimization regarding the service access mechanism would not make much of a difference regarding the overall system's performance if the execution of the visual language would incur in a high overhead, as typically process-based languages are executed by an interpreter. On the contrary, in this dissertation we propose to compile the visual specification of a process into executable code. One of the challenges of doing so is that the resulting code should still support the concurrent execution of multiple process instances. The choice of applying compilation to the execution of processes brings the following benefits. In addition to the potential for providing better performance through the optimization of the generated code, compiling processes also helps to simplify the design of the corresponding runtime system. As opposed to having a full-blown process interpreter, it is enough to design and build a flexible container of compiled processes.

Following this approach, in the last part of the dissertation we present the design of a flexible architecture for a process support system. Flexibility is an important aspect of our design which, according to our experimental results, does not contradict the goal of building an efficient system. First, flexibility enables JOpera to support heterogeneous types of services. To do so, plug-ins are used to map the invocation of a service to the corresponding protocol in the most efficient manner. Second, the flexible architecture of JOpera's kernel can be deployed in a variety of configurations. This way, costly features such as reliable process execution can be added only if they are truly needed. Likewise, the system shows good scalability when deployed in a cluster-based configuration, as large workloads are shared among multiple cluster nodes. Thanks to a wise choice of architectural abstractions, the code generated by the compiler is kept independent of the actual configuration of the kernel into which it is loaded. Third, flexibility is also a fundamental property for an autonomic system, where the optimal configuration is determined automatically at runtime.

Estratto

Questa dissertazione unisce argomenti di diverse aree di ricerca. Prima di tutto, proponiamo l'applicazione dei linguaggi visuali alla composizione di servizi. Al fine di connettere servizi di diversi tipi in un sistema di grandi dimensioni, le interazioni temporali fra di essi vengono definite con il JOpera Visual Composition Language. In confronto con la sintassi testuale (o basata su XML) dei sistemi esistenti, il linguaggio proposto usa una sintassi grafica molto semplice. Questo linguaggio viene usato per specificare il flusso di dati e di controllo che attraversa le varie chiamate ai servizi. In questo modo, a partire da un insieme di servizi riutilizzabili è possibile costruire rapidamente applicazioni distribuite attraverso il disegno delle loro interazioni. Per ottenere ciò, presentiamo come caratteristiche quali la scalabilità visiva, il posizionamento automatico degli elementi di un grafo, e multiple viste abbiano influenzato il progetto dell'ambiente grafico per la composizione di servizi del sistema JOpera. Per poter applicare il sistema a esempi realistici, il linguaggio è stato completato con lievi modifiche aggiungendo costrutti quali ricorsione, iterazione e riflessione. Grazie al sistema JOpera, il linguaggio visuale per la composizione di servizi è stato applicato a molti ambiti, come documentato dagli esempi inclusi nella dissertazione.

Il nostro metodo visuale per rappresentare la composizione di servizi è basato sul concetto di processo. In questa dissertazione prendiamo a prestito la nozione di processo di business per estenderla alla descrizione delle architetture orientate ai servizi. Di conseguenza, la struttura di un processo definisce l'ordinamento parziale delle chiamate ai suoi servizi, gli scambi di dati fra di essi e il comportamento in caso di problemi o condizioni eccezionali. In questo contesto, un contributo importante della dissertazione consiste nell'idea che un linguaggio di composizione debba essere ortogonale rispetto ai tipi di componenti usati. Più precisamente, nel sistema JOpera la composizione è definita al livello delle interfacce dei servizi. Quindi un processo è completamente indipendente dai meccanismi e protocolli usati per accedere all'implementazione dei suoi servizi. In altre parole, il nostro linguaggio di composizione non è limitato a descrivere come componenti di un tipo particolare (ad esempio, Web services) debbano essere composti. Invece, abbiamo generalizzato il concetto di servizio astraendo le caratteristiche comuni a molti tipi diversi di componenti in un meta-modello aperto. Questa astrazione ha diverse implicazioni importanti. Con la possibilità di usare un insieme aperto e ampio di tipi diversi di servizio, il linguaggio di composizione è più semplice perché molti costrutti (ad esempio, la rappresentazione di una invocazione sincrona o asincrona) possono venire spostati dal linguaggio di composizione al meta-modello dei componenti. Inoltre,

il compositore di servizi è libero di scegliere il meccanismo più appropriato per accedere alla funzionalità di un servizio pre-esistente. Quindi, durante l'esecuzione, il costo di accedere a un servizio puó venire minimizzato, in quanto diventa possibile scegliere il meccanismo di accesso più efficiente.

Questa ottimizzazione sul costo di accesso ad un servizio non avrebbe un grosso impatto sulle prestazioni globali del sistema se l'esecuzione del linguaggio visuale fosse inefficiente, come accade tipicamente nel caso dell'esecuzione interpretata dei linguaggi orientati ai processi. Al contrario, in questa dissertazione si propone di compilare la specifica visuale di un processe in codice eseguibile. Una delle difficoltà in questa soluzione consiste nell'ottenere del codice che permetta di gestire l'esecuzione di più di una copia di un processo alla volta. Tuttavia, la scelta di applicare tecniche compilative all'esecuzione dei processi porta i seguenti benefici. In aggiunta al potenziale di fornire prestazioni migliori attraverso l'ottimizzazione del codice generato, la compilazione dei processi facilita la semplificazione della struttura del sistema di esecuzione corrispondente. Invece di dover costruire un interprete, è sufficiente preparare un contenitore flessibile di processi compilati.

Seguendo questa soluzione, nell'ultima parte della dissertazione si presenta il progetto di una architettura flessible per un sistema di supporto ai processi. La flessibilità è un aspetto importante del nostro sistema che, secondo i risultati sperimentali inclusi, non contrasta con l'obiettivo di costruire un sistema efficiente. La flessibilità infatti permette a JOpera di usare un insieme eterogeneo di tipi di servizi. Per fare ciò, dei plug-ins vengono utilizzati per trasformare nel modo più efficiente la chiamata di un servizio nel protocollo corrispondente. Inoltre l'architettura flessibile del sistema JOpera puó venire adattata a diverse configurazioni. In questo modo, caratteristiche costose come l'esecuzione affidabile dei processi possono venire aggiunte solo se veramente necessarie. Allo stesso modo, il sistema presenta una buona scalabilità quando viene distribuito su un cluster di computer, in quanto grossi carichi di lavoro vengono condivisi dai diversi nodi del cluster. Grazie a una buona scelta di astrazioni architetturali, il codice generato dal compilatore rimane indipendente dalla configurazione del sistema in cui viene eseguito. Per finire, la flessibilità è anche una proprietà fondamentale per un sistema autonomico, dove la configurazione ottimale è determinata automaticamente e dinamicamente.

1. Introduction

1.1. Motivation

Composition is a well established standard engineering practice, whereby development proceeds by recursively assembling a set of pieces designed to fit together.

Software systems are also increasingly built in a similar way, or at least they should according to many sources (e.g., [214]). In fact, since the very early beginnings of the information age, using components to build software systems was deemed a promising idea, to be quickly followed by a blossoming software component industry whose reusable pieces of software could seamlessly be integrated together to build useful applications [154].

Along this direction, in the past forty years, a very large body of research on software components, components models, component based software engineering and the like has been produced. In practice, the component based frameworks that have been developed (such as Delphi [128], COM, CORBA, EJB, and many others) have also been quite successful as relatively well developed marketplaces of reusable software components have been established within the boundaries of a given component model [229].

Given the current level of development, where the Web service paradigm has been recently introduced to address interoperability issues across heterogeneous component models, in our work we have chosen not to describe yet another component model to avoid that our approach to composition would be limited to that particular component model.

Instead, we shift the focus from components to services, i.e., "software components with no strings attached" [225]. Moreover, in this dissertation we develop a language and a system for service composition, as we believe that component modeling is only half of the work, and that composition, i.e., defining in an executable way how services should be composed together, is equally important.

By keeping the definition of a service very general, we are able to propose a visual composition language that can be applied to model and execute the composition of many different types of services along the time dimension. To do so, the language is based on the notion of process, which describes the interaction between service interfaces in terms of data flow and control flow graphs. By relaxing the constraints on the types of services that can be composed we have kept our composition language both simple and general, as the complexity of modeling the invocation of heterogeneous component types has been pushed from the composition language to our

flexible component meta-model. One benefit of this approach is that adding support for additional types of services does not affect the definition of the composition language.

"A new language is not enough, unless there is also a compiler for it" [255]. Following this guideline, we have designed and built a whole set of tools to support the execution of processes defined with the JOpera Visual Composition Language. The JOpera system currently comprises a visual development and monitoring environment, a compiler targeting multiple process execution platforms, and a flexible runtime kernel that can be deployed in a variety of configurations.

Supported by the JOpera system, our language for service composition has been applied to several different application scenarios, as documented by the examples shown throughout the dissertation. Finally, we include a set of measurements to motivate and validate our approach.

1.2. Contributions

This dissertation brings the following contributions to the field of process-based service composition.

1. A visual language, as opposed to an XML syntax, should be used for programming process-based service composition.

   Software composition can be a great application domain of visual languages, as a two dimensional syntax can represent quite well non linear interactions between a set of services. In this dissertation we define a new visual language, the JOpera Visual Composition Language, with a very simple syntax, which is however powerful enough to be applied in realistic settings. Its main innovations are:

   - Processes are programmed mainly (but not only) by drawing a data flow graph linking input and output parameters of service invocations.
   - To address visual scalability issues, the control and data flow graphs of a process are displayed and edited separately. Furthermore, multiple views over the same data flow graph are supported.
   - Service interface adaptations that require XML data manipulations can be specified visually with the same syntax used to compose the mismatching services.
   - Iteration is supported through list-based split/merge operators, explicit control flow loops and recursion.
   - Reflection, through system parameters and system services, is used to model the interaction of a process with its environment and provides support for dynamic adaptation of processes and late binding of service interfaces to their implementation.

2. Composition and Components should be kept orthogonal.

   The JOpera Visual Composition Language defines composition at the level of service interfaces. The actual service invocation mechanism is intentionally kept transparent, as far as the definition of the processes is concerned. Given the wide range of existing component models and the corresponding service invocation mechanisms, limiting composition to a particular type of component is unnecessary, as the developer cannot choose the most appropriate one in terms of performance, reliability, security, convenience and ease of use. In other words, we believe that constraining composition to Web services only is a big limitation, as there are many existing, alternative, and established types of service access mechanisms that could be used, depending on the boundary conditions.

3. Processes which define how services are composed should be compiled for execution.

   In most existing systems, process models are interpreted while they are executed. We believe that visual, process based tools will not reach widespread acceptance if they cannot deliver a level of performance which is comparable to traditional programming languages. In JOpera, by defining a visual composition language, we believe we offer an interesting alternative, as far as the usability towards rapid composition is concerned. Furthermore, in order to achieve efficient execution, the processes are compiled to Java executable code. This code is dynamically loaded into JOpera's runtime kernel and is used to manage the execution of multiple concurrent instances of a process.

4. A highly flexible architecture for a process support system.

   Flexibility is a key property of JOpera's architecture both in order to support the invocation of services of a heterogeneous set of component types and to enable the deployment of JOpera in a wide variety of configurations. In these two aspects, flexiblity would seem to reduce the efficiency of the system, one of the reasons why we choose to compile processes. However, flexibility is useful to support the choice of the most efficient invocation mechanism for each component type. Furthermore, it also allows system administrators to create a configuration of the system with only the necessary features. For example, thanks to a flexible architecture, the user is empowered to make the most appropriate trade off between reliability and performance. Likewise, flexibility is an important characteristic of a dynamically reconfigurable system, where the optimal configuration is determined autonomously based on the current workload.

1.3. Structure

This dissertation is organized in two parts. In the first we define the visual language and the underlying model for service composition; in the second part we present the

system used to develop, compile and run the composite services.

Chapter 2 presents related work on various research areas (Visual Programming Languages, Software Composition – including Web Service Composition – Process Modeling Languages and Process Management Systems) touched by this dissertation.

Chapter 3 defines the JOpera Visual Composition Language, a glue language to draw connections between services and visually specify their interactions along the time dimension. The language is very general as it makes very little assumptions about the nature of the services to be composed and provides a simple, graph-based visual syntax which complements quite well existing XML-based approaches.

Chapter 4 is about the JOpera Component Meta-Model. After defining how to model different types of services (e.g., Web services, UNIX applications, Java classes, and so forth) we give many examples on how to use them as components within a process.

Chapter 5 describes the Opera Modeling Language, our contribution related to process modeling languages. This language, based on XML, is the internal storage representation of the processes developed using the visual syntax defined in Chapter 3.

Chapter 6 discusses how to execute the processes. As opposed to traditional approaches, where the process models are interpreted by an execution engine, in JOpera we introduce a compiler which generates executable Java code which uses the runtime facilities provided by the rest of the JOpera platform.

Chapter 7 presents the design of a radically new architecture for a process support system. The design of the JOpera process execution kernel attempts to find an optimal point between flexibility and efficiency as the system can be deployed in a variety of configurations to fit with the given reliability and scalability requirements.

Chapter 8 reports some interesting experimental results about the performance of critical parts of the system.

Chapter 9 summarizes the contributions of the thesis and discusses future research directions.

Note: in the first two chapters we have interleaved several examples on how to apply the ideas presented in the surrounding text. Although they can be skimmed on a first read, we believe that the exercise of creating a new language is not complete without both showing how to apply it in realistic settings as well as building a set of tools to support it [255].

2. Related Work

This dissertation brings together ideas of different research areas. Our work is related to both visual languages and component based software engineering, since we are interested in visually building applications and systems out of reusable and composable parts [229]. However, instead of focusing on typical composition issues regarding how the "spatial" architecture of a software system can be specified in terms of components and connectors [2, 156], we have focused the JOpera Visual Composition Language on describing how services should be composed in "time" [91].

In this chapter we will attempt to explain our views regarding the difference between services and traditional components and why when composing services it becomes important to model their interaction in the temporal dimension. To do so, in our approach the notion of service composition is closely related to the one of *process*, as it originally appeared in the workflow management community [83]. Very recently, several business process modeling and enactment tools have evolved into mega-programming [254] environments based on the service composition paradigm [42].

In this dissertation we have not only designed a visual, process-based, service composition language, but we have also built a system supporting it, featuring both a visual development and monitoring environment for rapid service composition as well as a flexible process runtime execution kernel. In this aspect, our work also improves the state of the art in process execution engines, as we present a flexible architecture for a process management system, which can be tailored to different levels of performance and which uses a compiler – as opposed to an interpreter – as a mean to achieve efficient execution of the visual language.

2.1. Visual Programming Languages

Starting with the pioneering work of the SKETCHPAD system [223], visual languages and tools have been used with success for many different purposes (e.g., programming [126], user interaction [196] and visualization [231]). Visual languages attempt to provide an effective, graphical, non-linear representation which has been applied with success to modeling (e.g., UML [197]), parallel computing [25, 32], laboratory simulation [242], image processing [226], workflow description [256], hypertext design [45], and even object-oriented programming [53, 114]. It is now widely recognized that a visual language is not better than a textual representation *per-se*, but – as with every kind of tool – a graphical notation may be more (or less) useful depending on the context [191]. Similar to [167], in this dissertation we show

that also software *composition* can be a good application domain for a graph-based, visual notation.

In particular, we reuse the notion of data flow [62], as a representation of the interactions between service invocations. For it provides a simple and intuitive notation, the data flow paradigm has been used by many existing visual languages [104]. However, this simple, side-effects free representation requires to be extended with additional constructs to be applied in practical settings. In the past, there have been many contributions concerning the problem of extending data flow languages with *iteration* constructs. A survey can be found in [166], while an example of iteration through vector operators and conditional switches is [17]. Similarly, *reflection* [151] is an important feature of a composition language. With it, the visual syntax is extended to model the interaction between a program and its environment (Section 3.7). By using terms such as "higher order functions", similar ideas have been applied to data flow based visual languages in the past [79].

In addition to describing the data flow structure of the interaction between different services, in the JOpera Visual Composition language we have also included a separate description of their control flow dependencies [187]. In the past, many graphical formalism have also been developed in this area. Here we mention some contributions that have been applied to workflow modeling. Examples include State Charts [100], used in the the Mentor project [257] to achieve distributed execution of the various workflow steps, or Petri Nets [190, 234] and variations such as Object Coordination Nets (OCoN) [256]. These formalisms have a natural visual representation, which provides the user with a good overview over the partial order of invocation of the services.

Nevertheless, when applied to service composition, one of the limitations of a visual language based only on control flow concerns the lack of a visual notation for specifying adaptations between mismatching service interfaces [15]. To this end, many different domain-specific visual tools and languages have been proposed. Mapforce [11] is a commercial data mapping tool for visual data integration between heterogeneous XML and database sources, which can also generate Java, C#, C++ executable code and XSL transformations [243]. In [192], the Visual XML Transformer (VXT) language has been introduced, advocating the suitability of visual programming techniques to simplify the specification of XML data transformations. In it, a set of Visual Pattern-Matching Expressions are used to generate the corresponding XSL transformation. Likewise, in [268], a visual formalism has been applied to the definition of the structure of an XML document and, augmented with a graph rewriting mechanism, used to specify document transformations. Another form of data transformation is provided by visual query languages [43], which have also been applied to XML documents (e.g., Xing [71], XML-GL [44]). Unlike in JOpera, where the same visual language is used both for modeling the composition of services as well as for specifying the necessary adaptations, these languages and tools are focused only on describing XML data transformations. Mismatching service interfaces were already a problem in the pre-Web services era. In the context of Electronic Data Interchange (EDI) systems, a visual language and environment

for EDI message translation has been presented in [94]. Similar to JOpera, the mappings, which are compiled to a different representation for execution, can still be interactively debugged using the same visual notation.

2.2. Software Composition

The idea of developing large scale applications by composing coarse grained, reusable software component modules has been pioneered by [154]. In [254] the term *megaprogramming* has been proposed to describe the construction of large scale software systems by composition of – so called – megamodules [188]. It has also been widely recognized that *composability* is a valuable property of a software system [158]. As opposed to closed solutions, open, composable systems can foster network effects thanks to their potential for reusability [48].

In the past, the idea of component based software engineering [101] has surfaced many times as the next silver bullet [31], which would be expected to revolutionize the software industry [52]. On the one hand, the full potential of component based software engineering has not yet been reached. For example, concerning the quality, or the lack thereof, of current software components, the notion of *trusted* components has been recently brought forward [160]. On the other hand, several different complementary (and competing) component-based frameworks have appeared targeting specific programming languages, platforms and application domains [69] (such as Delphi [128], CORBA [177], J2EE [56], COM [28] and many others [54, 180, 232]).

As listed in [224], in the literature there have been many definitions of the term "software component", each with its own architectural assumptions and the corresponding reusability constraints. In general, [16, 199] characterize and classify software composition systems by their component model, composition technique and composition language.

1. The *component model* defines how to describe components, and includes rules for exchanging equivalent ones. Abstraction, modularity and information hiding are all important features of a component model [184]. Furthermore, standardization at the level of the component interfaces should enhance their reusability and substitutability, while lowering the learning curve [149, 159].

2. The *composition technique* describes the mechanisms used for composing the components, with the ability of defining parameter types and specific data exchange protocols. Furthermore, to increase reuse, these techniques should provide support for adaptation (to fit a component to a given interface) and gluing (to mediate between different components) [195].

3. The *Composition language* influences the way composite systems are specified. It defines how to describe the architecture of a system built out of components, e.g., in terms of different styles [211]. In [82], it has been proposed that composition (or "glue") languages, should be separated from programming languages, which are instead more useful for implementing the functionality of

> the individual components [203]. A framework to classify and compare many existing architecture description languages has been developed in [156]. These languages with the appropriate supporting tools for active specification should be the basis for a composition-based software construction process. Since software systems tend to evolve and grow over time, the languages used to describe them should support the corresponding evolution of their architecture. Finally, a composition language should also lend itself to composition, i.e. by providing modularity constructs.

In this context, [225] gives a modern perspective on the relationship between component based software engineering and service oriented architectures. A *service* can be seen as a kind of *component*, as individual services can and should be composed into larger systems [183]. As opposed to traditional software components, the reusability of services is greatly enhanced, because

> a service is an instantiated configured system that is run by a providing organization. That is, a service is fully grounded. Ultimately, it includes the power supply to the server machines as well as the organization that somehow manages to pay the power bill [225].

Therefore, when composing systems out of reusable services, the static, "spatial" relationships between the component services become less important, due to the fact that all of the dependencies of a service – by definition – have been taken care of by the providing organization. Moreover, the interoperability between the services is guaranteed by the standardization of the mechanisms and protocols used to interact with them [5]. Instead, it is more useful to define composition in terms of the dynamic interaction of services along the "time" dimension [91].

Following these ideas, with the abstraction of *service*, i.e., an interface linked to a set of access mechanisms, in Chapter 4 we generalize the notion of component to enable composition across heterogeneous component models, i.e., not limited to a specific access mechanism, such as Web services [245, 246]. Likewise, composition across heterogeneous component models has also been advocated in [174], where the feasibility of integrating EJB with COM components has been demonstrated in the Vienna component framework [175].

2.2.1. Web service composition

Although they may not solve all component integration problems [236], emerging Web service technologies show great promise in reducing the complexity of interconnecting heterogeneous software components distributed across the Internet [61, 73]. They provide standard protocols for invoking (SOAP [245]), describing (WSDL [246]), and discovering (UDDI [173]) services in a platform and vendor independent manner [89]. Web services collaboration has been named the "Next Big Thing" [269] because Web services can realize their full potential only through the ability to compose complex services out of agglomerations of basic ones [42].

More precisely, once it is possible to interact with individual services, the ability to reuse, compose and describe relationships between basic services becomes important [5]. Furthermore, a stateful Web service may export multiple operations, which may need to be invoked following a certain interaction pattern. Another innovative aspect of Web services consists in the flexible and dynamic means of assembling different services. To do so, services advertise their capabilities so that they can be automatically discovered by the clients composing them [65]. Considering that it is not realistic to expect clients to be able to interact with arbitrary services, current efforts focus on enabling alternative implementations of previously known services to be located and invoked [110].

To denote these ideas various terms have been proposed: choreography [248], orchestration [112], automation [227], coordination [113], collaboration [66], and conversation [249]. In our case we prefer the term *composition*, since we are interested in developing applications by composing existing and reusable building blocks [263].

Web service composition is a very active area, where many different projects and many systems (e.g., [5, 34, 42, 47, 143, 165, 183]) are currently under development spanning from the extension of traditional programming or scripting languages (e.g., [12, 107, 121, 161, 221]) to new, ad-hoc languages (e.g., XL [74, 75]), including XML-based process modeling languages [29, 66, 112] as well as visual programming languages [167] and data-driven modeling languages [45].

Concerning the limitations of traditional programming languages when applied to coarse-grained composition, already in [82] the case for a separate "glue" language to coordinate the individual components was presented. Furthermore, referring to the old impedance mismatch problem between programming languages and databases [51], it has been argued that a similar problem exists with Web services [74]. Although more and more tools (e.g., [12, 107, 161, 221]) are being developed to address some of these issues, interacting with such coarse grained units of compositions by exchanging complex XML documents is still cumbersome to do with ordinary programming languages.

Our alternative approach towards a language for composition at a higher level of abstraction originates from the workflow area [83], where process modeling languages and related tools have been evolving to support the composition of Web services [141, 269]. In fact, both the emerging Business Process Execution Language for Web Services (BPEL4WS [112]) and the competing Business Process Modeling Language (BPML [29]) specifications use an XML-based syntax to represent how the Web services are composed into executable processes.

It should be noted that some efforts are currently concentrated in automating some of the tasks involved in composing services residing at different location and platforms by leveraging semantic annotations in their interface descriptions (e.g., [155]). Hence, an XML syntax appears to be well suited for supporting automatic service composition. Nevertheless, we would like to emphasize that no matter whether a Web service composition has been manually constructed by a human programmer, or matching services have been connected automatically using additional semantics, a *visual surface language* (such as the JOpera Visual Composition Lan-

guage), which can be used to give a complementary, visual representation of the result, is of fundamental importance to enable its understanding.

2.3. Process Modeling Languages

As previously mentioned, our approach to modeling service composition is based on the notion of *process*, which is related to the term workflow. The Workflow Management Coalition (WfMC) originally defined *workflow* as

> the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules [135].

Apparently, this definition restricts the scope of applicability of processes to business automation scenarios. This assumption can be considered as a result of the evolution of workflows from the groundbreaking work on office information systems [67, 270] and related studies on organizations [172]. Nowadays, Business Process Management Systems (BPMS) are defined as

> a generic software system that is driven by explicit process designs to enact and manage operational business processes [239].

Still, we could remove the word "business" to have a more general definition with a wider range of applications. In any case, such definition implies that a modeling language is required to specify the processes to be enacted by the underlying "process-aware" system. As opposed to hard-coding assumptions about a business process into the architecture of a corporate information system, having an explicit description of a process has several advantages:

1. Explicit workflow models enable actors to track the progress of active processes and perform off-line analysis of the executions which can provide useful feedback for improving the performance of the processes and contribute to the efficiency of an organization. Thus, in addition to task coordination issues, process models typically include a specification of non-functional aspects, such as deadlines, priorities, as well as resource constraints.

2. A workflow model represents the high-level procedural aspect of a business process. It can be used as documentation for business analysts and internal auditors, but – as opposed to other modeling notations (e.g., UML [63, 256]) – still retains formal, executable semantics that can be automatically enforced by a workflow engine.

3. Likewise, if the process of modeling is essential in creating shared understanding in an organization, the modeling technique employed is meant to keep discussions on the right track, and should be chosen accordingly [123].

4. In addition to information and data, also business processes, i.e., the connection of tasks in a value chain, are valuable assets of an organization [209]. Thus, similar to database management systems, which are normally used for the safekeeping and management of an organization's data, also process management tools should be used to model, analyze, and execute its business processes [215].

In the past ten years, a very large number of process modeling languages have been proposed both from the industry and academia. (e.g., [26, 64, 83, 164, 233, 239]). Process modeling languages have been applied to several domains, including business process modeling [146, 200], e-commerce [6, 202, 235], virtual laboratories [4], DNA sequencing [157], scientific computing [153, 252], grid computing [22, 39], and software development [78, 181]. More recently, the notion of process-based service composition has appeared [42, 147]. To address some of the requirements of these areas, it was suggested to extend such languages with features such as flexibility [237], event-based interaction [41], and transactional properties [148, 206].

One of the contributions of [98] was the abstraction from the variety of existing languages of a set of common constructs and features into a *canonical* representation for processes. By defining mappings from different, domain-specific representations, the same, generic process model can be used to execute processes belonging to different application domains. In this dissertation we build on this idea, as we have structured our process modeling language across different levels and defined mappings across each of them. As shown in Figure 2.1, each level corresponds to a particular function that determines the characteristics of the language.

- At the user-oriented level, processes are displayed using a visual notation (Chapter 3). The same notation is used both at development and at debugging, monitoring time.

- However, a tool-oriented XML-syntax is used for the internal storage of the processes (Chapter 5). This facilitates the development of an open process development toolchain, where a set of editors, model checkers, compilers share a common representation optimized for efficient automatic processing.

Function	Syntax	Language
Display	Visual	JVCL (Chapter 3)
Storage	XML	OML (Chapter 5)
Execution	Platform-dependent	Java, BPEL, OCR (Chapter 6)

Figure 2.1.: *Summary of the process modeling languages presented in this dissertation*

	Workflow pattern	OML	BPEL [258]
1	Sequence	+	+
2	Parallel split	+	+
3	Synchronization	+	+
4	Exclusive choice	+	+
5	Simple merge	+	+
6	Multichoice	+	+
7	Synchronizing merge	+	+
8	Multimerge	-	-
9	Discriminator	+	-
10	Arbitrary cycles	+	-
11	Implicit termination	+	+
12	Multiple instances (without synchronization)	+	+
13	Multiple instances (with a priori design knowledge)	+	+
14	Multiple instances (with a priori runtime knowledge)	+	-
15	Multiple instances (without a priori runtime knowledge)	-	-
16	Deferred choice	+/-	+
17	Interleaved parallel routing	+/-	+/-
18	Milestone	+	-
19	Cancel activity	+	+
20	Cancel case	+	+

Table 2.1.: *Workflow patterns supported by the Opera process Modeling Language*

- Before they can be executed, processes are compiled to other representations such as OCR [98, 21], BPEL [112], and Java (Chapter 6). This approach is very similar to emerging Model Driven Architecture (MDA [127]) techniques, as a refined, executable representation of a process is generated automatically from its higher-level design.

Incidentally, a similar approach is currently followed by most modeling tools, where the visual UML representation is internally stored (and exchanged) using the XML-based XMI language. This way, different UML editing tools not only share the same visual language, but also achieve interoperability, as the diagrams produced with one tools can be read by another one.

Alas, this level of interoperability has not yet been achieved by current process modeling tools. In practice, given the number of process modeling languages and tools on the market, issues such as runtime interoperability and portability of process definitions become very important [60]. To address this technology *lock-in* problem [158], several major players are currently proposing process modeling

standards. Although the current leader is represented by the BPEL4WS specification, which we briefly present in the following section, a consensus has not yet been reached on a common process modeling language.

Different process modeling languages can be also compared in terms of their expressive power. More precisely, an evaluation based on so-called workflow patterns can be carried out [233]. To give an idea on how the Opera process Modeling Language (Chapter 5) fares in this regard, in Table 2.1 we have listed what are the control flow patterns that can be naturally expressed.

2.3.1. Business Process Modeling and Execution Language for Web Services

The Business Process Execution Language for Web Services (BPEL4WS, or BPEL [112]) is a process modeling language for Web service composition. It contains abstract and executable processes. Abstract processes are used for describing business protocols, while executable processes may be used to implement composite services. Based on an XML-syntax, BPEL supports a fixed set of basic activities (e.g., `invoke`, `send`, `receive`, `assign`) to represent the synchronous or asynchronous invocation of services or data transfers between the global variables of a process. Furthermore, it also includes complex activities (e.g., `sequence`, `flow`, `while`, `pick`) which are used to define the structure of the process in terms of its control flow.

Although this specification represents the current state of the art in process-based Web service composition, its standardization process has not yet completed and further additions and modifications are being discussed at the time of writing. Figure 2.2 attempts to put the evolution of this language into context. BPEL originated from the fusion of two existing (and quite different) languages. Its graph-based constructs (such as `flow`) have been inherited from the IBM's Web Services Flow Language [142]. The block-based constructs such as `while` or `sequence` come from Microsoft's XLANG [227].

The presence of alternative, overlapping and inconsistent constructs has made it a challenge to add features such as exception handling [58]. Furthermore, although the language originates from the fusion of two different ones, it provides limited support for a large set of established workflow patterns [258] (Table 2.1). In practice, the language makes it particularly difficult to compose services with mismatching interfaces, as one of its underlying assumptions is that the services to be composed have perfectly matching interfaces [116]. Lately, as some more limitations have become apparent[1], the proposal of extending the language with support for including Java snippets was brought forward by IBM and BEA Systems [111]. Although the need for such an extension is clear, one may argue that, as opposed to Java, a .NET compliant language should be chosen instead. Thus, a technology which was originally tied to platform neutral Web services, becomes once again tangled into portability issues [216]. BPEL has also been criticized for lacking a clear formal underpinning.

[1] See Section 4.1 for a discussion on the limitations of restricting composition to Web services.

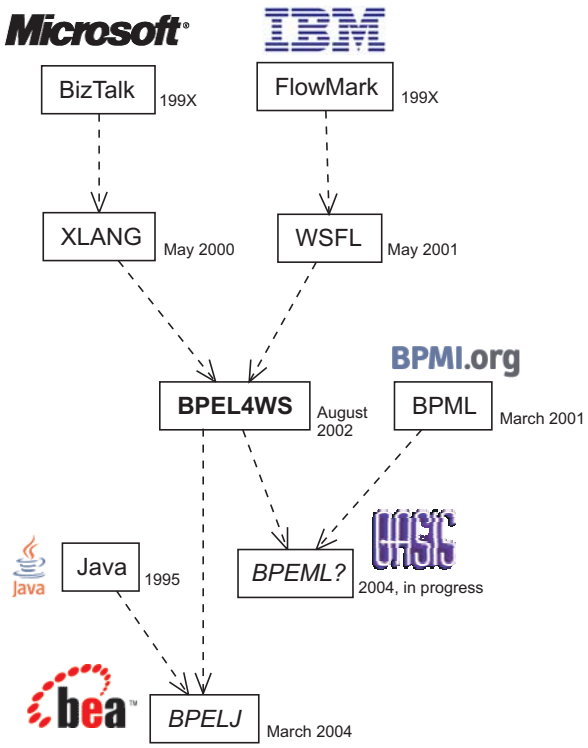


Figure 2.2.: *Evolution of the BPEL4WS specification*

> Although there are well-established process-modeling techniques combining expressiveness, simplicity, and formal semantics (such as Petri nets [190] and process algebras [162]), the software industry has chosen to ignore these techniques. So, the world is confronted with too many standards, mainly driven by concrete products or commercial interests. The only way to stop this is to ignore standardization proposals that are not using well established process-modeling techniques. This will force vendors to address the real problems rather than create new ones [236].

As we will show in the first part of the dissertation, the languages we developed share with BPEL the notion of process-based composition. However, there are many important differences that should be pointed out. First of all, *simplicity* was one of the goals in defining JOpera's process based language. Therefore, as opposed to creating a language by accumulation of features from existing ones, we purposefully kept the number of redundant (and arbitrary) constructs to a minimum. This approach helped both to lower the language's learning curve and to simplify the design and significantly reduce the development effort of the supporting tools. Furthermore, the JOpera Visual Composition Language does not use an XML-based syntax (Chapter 3). As we will show in Example 4.3, in addition to service composition, the same visual syntax can be also applied to specify interface adaptations. Furthermore, the process-based service composition language we have developed is not tied to a particular service access technology. In other words, Web services are

only one of the various types of components that can be composed into a JOpera process (Chapter 4).

To the best of our knowledge, the BPEL4WS [112] specification is currently supported by three implementations. In all cases the execution engines are meant to be deployed inside an application server. The Collaxa BPEL Server [49] is the most advanced as it comes with a graphical process designer and debugger. The visual notation employed has a very close mapping to the underlying BPEL document. This has the advantage that a BPEL document doesn't need to be edited at the XML level. On the other hand, unlike the JVCL language, the notation is not abstract enough to be applied to other process modeling paradigms. The second implementation is the Business Process Execution Language for Web Services Java Run Time (BPWS4J [106]) from IBM, which also includes an editor with minimal visual support. The third system supporting the BPEL specification is OpenStorm's Service Orchestrator [179]. In addition to a two-way graphical/XML editor, it features a runtime environment which can be deployed in both Java and .NET application servers.

2.4. Process Management Systems

Although the exercise of defining a new process modeling language is not too difficult, more work is required to actually build a system for the execution of such language. Thus, relatively less work can be found about distributed architectures for scalable process execution [92]. More specifically, scalability has been a common goal to be achieved through different means: replication at the database layer, distribution in the process execution engine and decoupled communication through events notification. Only rarely all of these approaches have been followed within the same system.

The idea of building a distributed workflow enactment system based on event communication and event-condition-action rules has been also proposed, e.g., in the EVE project [85] and the ORCHESTRA process support system [57]. The exchange of event notifications plays an important role in our approach. However, in our experience, ECA rules are only a useful intermediate representation to bridge the gap between graph based models, which can be more readily understood by the user designing the process, and the corresponding executable code (Chapter 6).

The theme of enhancing the system's fault tolerance and scalability through replication at the database layer has been pioneered by [124]. Also in the MOBILE project [102], in order to replicate the process execution layer, a scalable strategy for distributing the process data among separate databases has been proposed [208]. Although we compare the performance of a centralized, persistent repository with a distributed, volatile implementation, in Chapter 7 we do not pursue replicated storage any further.

Decentralization has been pursued by the MENTOR project [257], where process definitions are analyzed and automatically partitioned among distributed execution sites in order to avoid the bottleneck of a centralized engine [168]. This approach

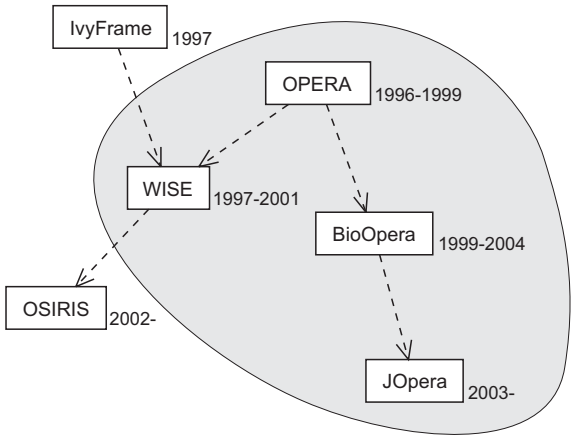


Figure 2.3.: *Evolution of the* OPERA *system.*
The gray area groups the systems developed at the Information and Communication Systems Research Group.

fits well with the requirements of workflows spanning across multiple organizations. However, it is possible for one execution site to become a hot spot, when it is involved in the execution of a large number of processes. To deal with this problem, techniques such as meta-data replication based on publish/subscribe and the ability to partition process navigation among alternative service providers have been employed by the OSIRIS project [207, 251].

Once a distributed process architecture has been designed, load balancing, network congestion and quality of service guarantees become interesting options. In [122] a cluster-based workflow management system has been presented focusing on a quantitative comparison of two different load balancing strategies. In [20] simulations are used to study how different workloads influence the load of the network and thus, the scalability of the workflow engines in the context of several distributed architectures. In [95] extensive simulations are used to validate a composition model with quality of service guarantees based on service overlay networks.

2.4.1. About the JOpera project

JOpera is the visual, process-based service composition system of the Information and Communications Systems Research group at ETH Zurich [185] and corresponds to the latest development step of a series of process support systems which were prototyped in the past decade. To put this dissertation into a historical perspective, in this section we briefly present the foundation on which we have built upon.

The original OPERA system developed as part of [98] has since undergone several generations and evolutionary branches, some of which are shown in Figure 2.3. In [9], the idea of applying workflow management systems to an area wider than business process modeling and enactment was first explored by arguing that such systems could provide a platform for distributed processing over stand-alone systems and

applications. Based on this approach, an example application scenario related to geographical information systems was presented in [7]. Following these ideas, in order to show the benefit of making workflow modeling languages more and more similar to traditional programming languages, the introduction of features such as exception handling [97] and inter-process communication [8] was proposed. As we will present in Section 3.4.3, exception handling is also supported by the JOpera system in a similar fashion. However, in Section 4.10 we will show a simpler solution to model the asynchronous interactions of different process instances. Instead of extending the process model with additional constructs, we included basic *send* and *receive* primitives in JOpera's component library.

Later on, the kernel of the OPERA system was extended with transactional capabilities [206]. The result was the WISE system, a platform for creating virtual enterprises, tailored to the business to business electronic-commerce area [6]. In this project, a first visual representation of OPERA's textual process modeling language was introduced by integrating a graphical modeling tool called IvyFrame [115]. Compared to the JOpera Visual Composition Language, the visual notation of WISE – used to represent only the control flow dependencies between the tasks of a process – was based on Petri nets. As part of the project, the Ivyframe tool was extended to support the whole lifecycle of a process with development, simulation and monitoring features. The WISE system was applied with success within the maritime industry [136].

Quite different from electronic commerce and virtual enterprises, bioinformatics and virtual laboratories were the original application area of the BioOpera system [4]. In it, a heavily refactored version of the original OPERA kernel was augmented with resource management and scheduling capabilities [186]. This allowed us to show the feasibility of applying a process support system to cluster [23] and grid [22] computing scenarios. More precisely, in BioOpera the notion of process was applied to model complex distributed computations to be enacted over one or more unreliable clusters of computers. Reliability was achieved through a persistent implementation of the process execution kernel and through the ability of automatically rescheduling failed task invocations. Given the complexity of managing long-lived computations in such distributed environments, [21] showed the feasibility of an approach based on autonomic computing principles [109].

Part I.

The JOpera Visual Composition Language

3. JOpera Visual Composition Language

This chapter introduces the syntax of the JOpera Visual Composition Language (JVCL), describing the visual representation of processes and their data flow (Section 3.3) and control flow (Section 3.4) structure as well as more advanced constructs such as iteration (Section 3.5), visual comments (Section 3.6), and reflection (Section 3.7).

3.1. Motivation

Why a new visual process modeling language? As we have seen in the previous chapter, there have been already many contributions, both in the areas of visual programming languages and visual process modeling.

Visual programming languages, however, have been mostly oriented towards programming in the small, positioning themselves on a level of abstraction comparable with traditional programming languages, such as C or Java [104]. In this domain, it has become clear that two (or three) dimensional approaches suffer from visual scalability problems [35], where the usability of such tools and languages decreases as the size of the diagrams increases [191]. Only recently there have been some attempts to shift the focus to programming in the large, where the composition of coarse grained software components (or services) plays a more important role [167].

In this dissertation, we have designed a visual *composition* language, whose main application domain lies in describing of how services are composed together [187]. We believe this is a more viable application area for a visual language, where a non-linear, two dimensional syntax can be most appropriate. Furthermore, with our language and tools we have also made an attempt to address the visual scalability problem[1].

Visual *process modeling* languages have been mostly based on adaptations and variations of existing graphical notations and formalisms (e.g., Petri-Nets [190, 234] or State Charts [100, 257]). Also within the UML community, business processes have been usually modeled using Activity diagrams [256], for which the underlying semantics has been upgraded to Petri Nets in the current UML 2.0 proposals [116]. The strong point of all of these approaches lies in the accurate description of the

[1] See Section 7.2.2 on page 150 for more information on the usability features of JOpera's visual development environment.

control flow of a process, where a large number of constructs is devoted to describing the partial order of invocation of the services composing the process, in order to support various branching and synchronization patterns [233]. However, as these notations are applied to service composition, some limitations become apparent:

- In order to provide an executable description of a process built out of interconnected components, it is not enough to model its control flow, as the components typically exchange some kind of information between their invocations.

- Most existing visual process modeling languages do not use a visual syntax to program the data flow, which describes how data is transferred across component boundaries. As an example, in the syntax of the UML 2.0 activity diagrams profile for Web service composition, the data flow transfers between the activities representing service invocations are programmed with a textual syntax inside comments associated to control flow edges [116].

- Very little can be done with a pure control flow approach, as far as the visual modeling of the necessary adapters between mismatching service interfaces is concerned.

- The control flow and, when supported, data flow aspects of a process model are usually overlayed in a single diagram [63]. This approach leads to unnecessary clutter and, given the complexity of real business processes [36], may hinder the usability and the success of such visual languages and tools.

In the language we describe in this chapter, we attempt to address such limitations by modeling processes primarily by their data flow structure. This way, developers can define the composition of services by drawing connections between their interfaces and, in realistic settings, also visually specify the required adapters[2]. Nevertheless, the control flow structure of a process is still accessible, as it provides a useful overview over the content of a process and the order of invocation of its components, but it is not the primary (and only) feature of the language, as such information can be partly derived automatically from the data flow graph.

Finally, one of the goals that influenced the design of the JOpera Visual Composition Language was to provide a simple, intuitive – and executable – visual notation to support the rapid, user-friendly development of processes composed of reusable services. To avoid misinterpretation problems [96] we reduced the number of *ad hoc* constructs and extensions to a minimum, keeping the balance between the need for expressive features and the constraints imposed on the underlying JOpera runtime platform.

[2]See Example 4.3 on page 69 for an example on how to visually adapt mismatching service interfaces

3.2. Processes and Tasks

A process is composed of tasks, which can be either *activities* (simple tasks) or *subprocesses* (complex tasks). Activities represent the invocation of a service, while SubProcesses represent the invocation of another process. As shown in Figure 3.1, in the JVCL language a task is drawn as a box with its name inside. An activity box has a single border; boxes for subprocesses have a double border to indicate nesting. Furthermore, the name of the service to be invoked or the process to be called can also be displayed in the task box. If necessary, e.g., to reduce clutter, the user can decide to hide this additional information. Given the abstract nature of most services and to keep the notation as simple as possible, we have chosen not to use icons in addition to names to illustrate the tasks' operations [196].

The tasks of a process are linked by data flow (Figure 3.2) and control flow (Figure 3.3) dependencies, therefore the structure of a process can be programmed by drawing two directed graphs. The nodes of these graphs represent the tasks and their data parameters. The edges represent control flow or data flow dependencies.

3.3. Data Flow

The data flow graph defines how the data is exchanged between the parameters of the various tasks of the process. The nodes of the graph represent the process, its tasks and their parameters. The edges represent data flow transfers.

More specifically, tasks are associated with a set of input and output data parameters. Input parameters are used to pass data to a task about to be started. Output parameters contain the results returned from the task once its execution has finished. This property is visually represented in the data flow graph syntax, as the tasks are connected with incoming edges to their input parameters. Conversely, outgoing edges connect tasks to their output parameters. It should be noted that these edges are not removable, since there cannot exist a parameter box disconnected from its task. To complement the parameter's name, it is possible to show its type inside the same box. The user may choose to display this additional information, e.g., to resolve type mismatches.

Similar to tasks, also processes have input and output parameters. However, to improve readability by giving a higher degree of freedom for the graph layout, the parameters of a process are linked to two separate shapes representing the input and output interface of the process.

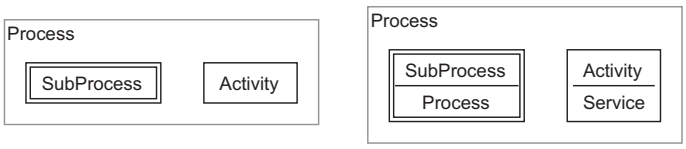


Figure 3.1.: *Syntax definition for the Activity and the SubProcess*

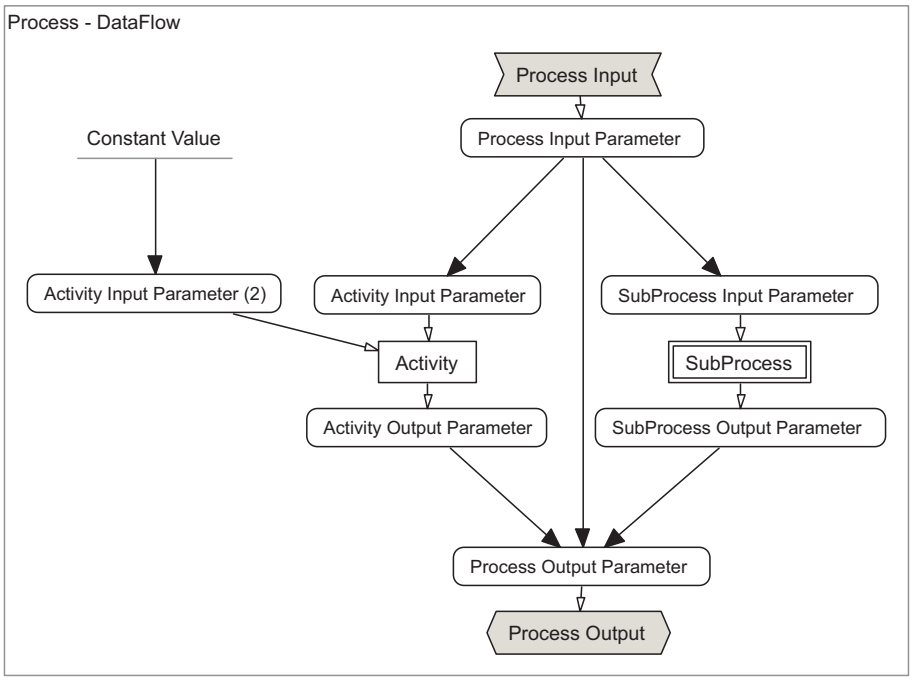


Figure 3.2.: *Data flow graph syntax*

3.3.1. Bindings

Data flow connections between parameters define how their content is transferred between them: a data flow *binding* is represented as an edge going from an output parameter box of a task to an input parameter box of another task. Furthermore, as shown in Figure 3.2, also constant values can be connected to input parameters of tasks.

The same parameter can be connected by multiple data bindings. For example, to copy data produced by one task to multiple ones, one output parameter box can be linked to multiple input boxes. Multiple incoming bindings are also allowed by using a *last writer wins* semantic: the value of the input parameter will be overwritten each time a task finishes and, at the end of the process, its value will be a copy of the output parameter attached to the task finishing last. This rule has been chosen considering that multiple incoming bindings are mostly used in loops or when the control flow merges from two or more alternative execution paths.

The same rule is also applied to the output parameters of processes. More specifically, if such a parameter is bound to a constant value or directly to a process input parameter, this binding is evaluated first, as the process is started. The remaining bindings are evaluated after their corresponding tasks have finished. Thus, the value of the process output parameter will be overwritten only if these tasks will have finished their execution, as specified by the conditions in the control flow graph.

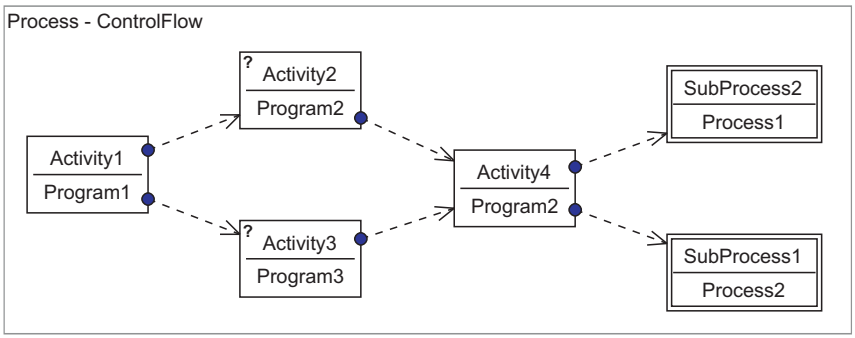


Figure 3.3.: *Control flow graph syntax*
Task boxes contain the name of the task and the name of the program (or process) to be invoked. `Activity2` and `Activity3` are marked with the condition icon.

3.4. Control Flow

The partial order of execution of the tasks inside a process is defined by its control flow graph, with tasks as nodes and control flow dependencies as directed edges. (Figure 3.3)

By definition, a data flow binding between two tasks implies a control flow dependency. This is because it is not possible to transfer data from task A to task B unless task A has successfully finished execution and B has not yet been started. It follows that a subset of control flow dependencies can be automatically derived from the data flow specification. Furthermore, extra control flow dependencies can be directly added to the control flow graph to model constraints in the order of execution of tasks that are not explicit in the data flow.

A control flow edge from node A to node B is used to show that task B cannot start until task A has reached a certain execution state associated with the edge. Examples of such states are: *finished* (by default), *failed* (when an error during the execution of the task is detected), *aborted* (after an user has killed the task), or *not reachable* (when the task has been skipped). The state is visually represented by the color of the dot positioned at the tail of the control flow edge. This makes it easy to follow, at runtime, whether a control flow dependency has been activated, as this only happens if the color of the task box, representing its state, matches the color on the edge.

3.4.1. Conditions

Start conditions, boolean expressions referencing parameter values, are associated to each task and can be used to model alternative execution paths. A task can only be started when all of its control flow dependencies are activated and its start condition is satisfied. Otherwise, if the condition evaluates to false, the task is skipped. In this case, to record this decision the state of the task is set to *not reachable*. Currently, start conditions are specified only in textual form as one of the task properties.

However, boxes of tasks with non-trivial conditions (e.g., `TRUE`) are marked with a small question-mark icon.

3.4.2. Synchronization

If there is more than one incoming control flow edge to a node C, it must be defined how the various dependencies are combined, as task C represents a potential synchronization point in the process where multiple execution paths merge.

By default, the semantic is to *and* all dependencies. For example, if there is a dependency coming from service A and another from B, task C cannot be started until both tasks A and B have finished. One exception to this rule is when there is a merge of alternative execution paths, in that case the semantic is to *xor* the connections. Similarly, for incoming connectors part of a loop in the graph, the semantic is to *or* the loop dependency with the others.

To provide a general way of modeling arbitrary synchronizations in the control flow, in addition to a condition, a task is associated with an *activator*, a boolean expression defining how to synchronize multiple incoming control flow edges. Such expression, normally generated from the control flow graph, can be also edited in textual form. In this case, the JOpera Visual Development Environment ensures that the graph topology and the activator remain consistent.

3.4.3. Exception Handling

Modeling failure handling behavior is an important requirement for a composition language, as exceptions are the rule when running processes in a distributed environment. As opposed to introducing an *ad-hoc* language construct, e.g., block-based exception handling, we extend the existing control flow graph construct as follows[3].

As shown in Figure 3.4, failure handling behavior is specified in the control flow graph by using connectors which fire on failure of a task. An exception handling task may be added to a process by drawing such connections from one or more tasks to it. Similarly, a compensation handler is added by connecting it to the task which may require compensation with a control flow connector which fires after the task is aborted. With start conditions applied to the output parameters of the failed task, it is possible to discriminate between different types of failures and activate the appropriate exception handler. By drawing an edge from the exception handler back to the failed task it is possible to retry its execution after the exception handler has finished. As an alternative, it is possible to resume the execution of the process along an alternative execution path, triggered by the failure.

With the previously described solutions, it is possible to handle the failure of individual tasks. Furthermore, in order to handle the failures of any tasks belonging to a certain part of a process, the same exception handler should be triggered by the failure of at least one task belonging to a certain set of tasks. To model this,

[3]It is possible to map a block surrounding a set of tasks to edges linking the tasks to the exception handler corresponding to the block [58].

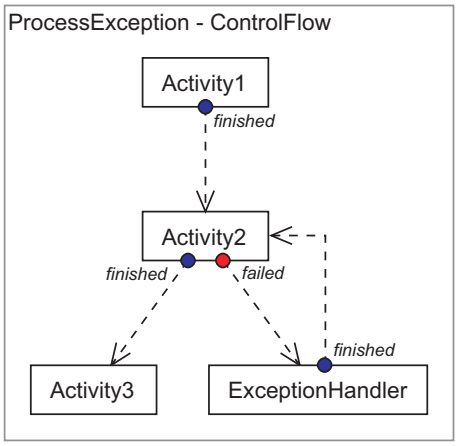


Figure 3.4.: *Control flow with exception handler*

it is possible to connect the various tasks to the same exception handler with the appropriate *failed* dependency and to select an *or* synchronization type between all of the dependencies. This way, the exception handler will be triggered by the failure of at least one of the tasks.

Another possibility, similar to block-based exception handling, is to attach an exception handler to a subprocess, so that it will be triggered by any failure occurring inside the called process. This is particularly useful when the exception handler cannot be added directly within the same process which contains the tasks that may fail.

Example 3.1: Book Prices

As a first example, we show how to use the basic features of the JVCL language in a Web service composition scenario where a process is used to compare the prices of books sold at various Internet stores. This process receives as input an ISBN number and returns as output an URL for a report containing the price comparison for the book. Since stores at different countries return prices in their own currency, the user may supply the currency to be used in the report as optional input parameter. The process contains the necessary steps to perform the currency conversion. The report also contains the book's author and title, retrieved from a library database, and a listing of the top 5 results returned by a web search engine looking for the author and the title of the book.

Process BookPrices

Figure 3.5 shows the control flow graph for the price comparison process. The process is composed of three activities (`Library`, `GoogleSearch`, `MergeReport`)

and one subprocess (`QueryBookPrice`). As its name suggests, `QueryBookPrice` involves contacting a book store to inquire about the price of a certain book identified by its ISBN. While this happens, the `Library` activity retrieves the author and title of the book. When the library query finishes, the web search is started and when all of the previous tasks are finished the report is generated.

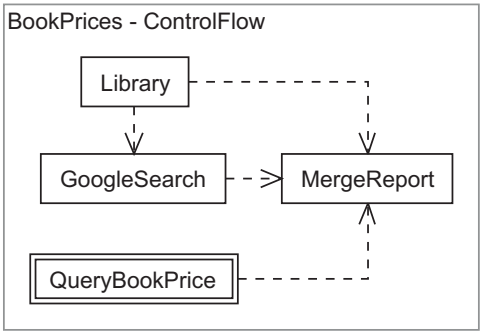


Figure 3.5.: *Control flow graph of the BookPrices process*

The data flow graph of this Process has been partitioned into two different views to enhance its readability. Figure 3.6 shows one view with data parameters and bindings of the `Library`, `GoogleSearch`, and `MergeReport` activities. While the second view in Figure 3.7 shows the data flowing through the `QueryBookPrice` subprocess.

The first view (Figure 3.6) shows one of the input parameters of the process (`isbn`) passed both to the `Library` and `MergeReport` activities. Given the `isbn` as input parameter, the `Library` activity returns the corresponding `author` and `title`. These two parameters are passed on to the `GoogleSearch` activity, which will run a web search using them as keywords and return the top 5 `results`. The `MergeReport` activity receives the `title`, the web search `results`, the `author` and `isbn` of the book, it uses it to generate a report and returns a `url` where it can be found. When the process is finished this value is returned as the `reporturl` output parameter of the process.

The rest of the data flow is shown in the view of Figure 3.7, which shows an example of the parallel split and merge iteration constructs presented in the following Section 3.5. In the example, they are used to simplify the process, because it can call in parallel four different services having the same interface with only one subprocess. Both `isbn` and destination `currency` process input parameters are passed to the `processQuery` subprocess, which also receives the identifier of the bookshop `service` to be called and the `source` currency of the price returned by the service. At runtime, a parallel copy of the `processQuery` subprocess will be executed for each element found in these two input parameters. In the example, the `service` and `source` parameter are bound to constants with a list of four strings, which contain service identifiers (`BooksCH, AmazonCOM, AmazonDE, BNCOM`) and the corresponding

currency identifiers (`CHF, USD, EUR, USD`). The prices returned by the parallel instances of the `processQuery` subprocess are merged into the `prices` input parameter of the `MergeReport` activity. Both views show the same data flow connection binding the output of the last activity with the output of the process.

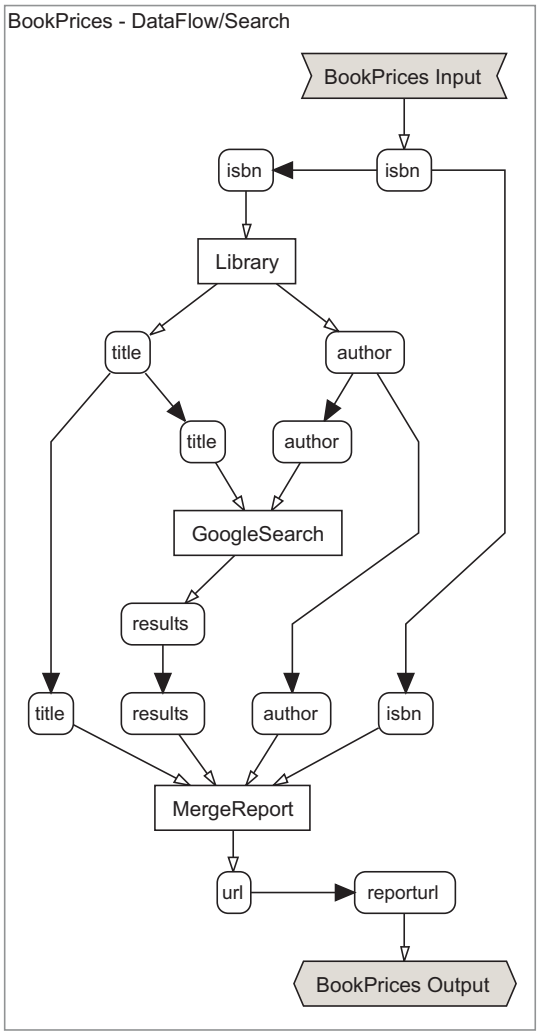


Figure 3.6.: *First data flow view of the BookPrices process*

Process QueryBookPrice

The `QueryBookPrice` process is called from within the `BookPrices` process. It contacts two Web services in order to inquire for the book's price and to convert it to the desired currency. Figure 3.8 shows its data flow graph. This process contains two activities: `QueryBookPrice, CurrencyConvert`. The input and output parameters of the process match the ones of the `processQuery` subprocess. The `isbn` of the book is passed to the `QueryBookPrice` activity.

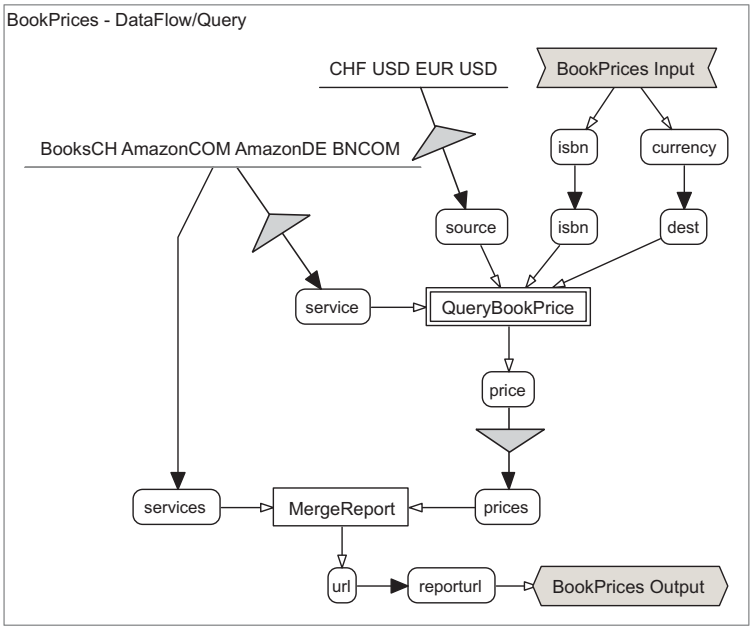


Figure 3.7.: *Second data flow view of the BookPrices process*

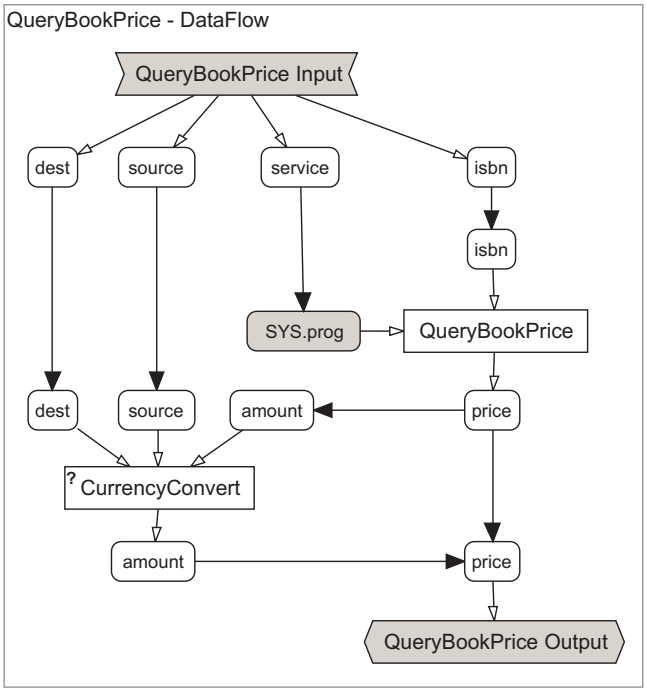


Figure 3.8.: *Main data flow view of the* `QueryBookPrice` *process*

In order to choose the services to call, the actual `service` name is assigned to one of the activity's system input parameters called `SYS.prog`, resulting in the invocation of the corresponding service. After the query has completed, the

resulting `price` and the `source` and `destination` currencies are passed to the `CurrencyConvert` service, which will return the corresponding `amount`. When the process finishes, the converted `price` is returned to the caller. It should be noted that the `CurrencyConvert` service is not invoked when the currencies are the same, in this case the price is returned directly from the result of the query.

3.5. Iteration

Supporting iteration in a language based on the data flow paradigm requires to introduce some auxiliary construct [166]. In the JVCL we rely on three constructs with a different degree of generality. First, we introduce two special data flow connectors used to repeat the same operation on every element of a list. Second, we have been experimenting with arbitrary loops in the control flow graph. Third, recursive subprocess calls are also supported.

3.5.1. List-based Loops

List-based loops can be used to repeat the same operation on a given set of values. When no data dependencies hold between the values, the operation can be performed in parallel. Otherwise, the task must be applied sequentially on each value. To achieve this, we introduce a pair of special data flow connectors, called *split* and *merge*. As in other graph rewriting schemes [25], the overall effect of these operators at runtime is to replicate a task node for each value of the input parameter list. This construct has also been classified as a *multi-instance* based workflow pattern, where the number of multiple instances of tasks is known in advance, before the first one is started [233].

This pair of operators has been originally introduced to support the modeling of data parallel computations, where a potentially very large workload can be subdivided in a number of small, independent partitions to be executed in a distributed

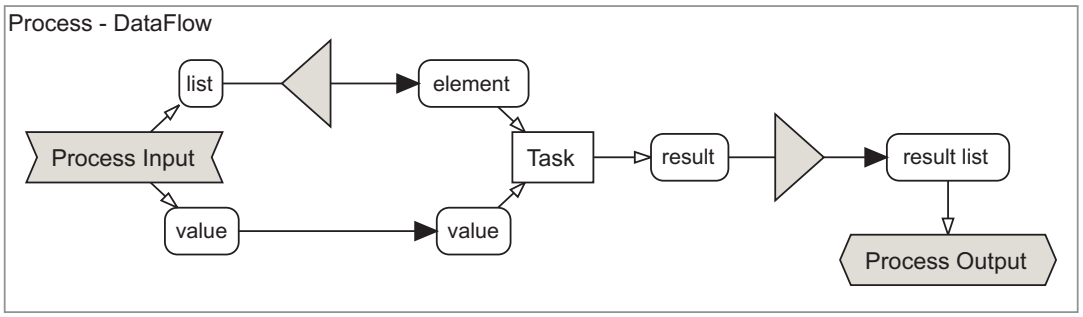


Figure 3.9.: *Data flow syntax of the list-based loops*

environment, such as a cluster of computers [4]. Especially if the amount of work is known in advance, the list-based loops offer a straightforward way of connecting the three steps of such computation: 1) the partitioning of the input data; 2) the parallel processing of the pieces; 3) the aggregation of the results, once all pieces have been completed.

Figure 3.9 shows how these operators are visually represented. In it, a `Task` is invoked for each `element` of a `list` producing the corresponding `result`. Such split operation is represented by the gray triangle on the data flow binding linking the `list` to the `element` parameter. Although each task receives a different `element` of the `list`, all tasks are invoked with the same `value`, as this parameter is connected with a plain data flow binding. Upon completion of all invocations, the merge connector is used to concatenate all results into the `result list` parameter.

By setting properties associated with the operators (Figure 3.16), the user can control whether the invocation of the tasks happens sequentially or in parallel and how the elements are extracted from the list. For example, JOpera can interpret a string with multiple words separated by blanks as a list of words. Similarly, JOpera can also split and merge arrays encoded in the SOAP protocol using XML tags as element separators.

In case of the parallel invocation of the multiple task instances, it is possible to control both how the failures that may occur during the processing of some of the elements are aggregated and how the parallel tasks are synchronized. In some scenarios, it may be useful to ignore some of the failures, as long as some of the elements can be successfully processed. Similarly, only if the results of all tasks need to be merged, it is necessary to wait for all parallel tasks to complete. Finally, in the case of a sequential split connector, the appropriate control flow dependencies between each task of the sequence are automatically inserted when the loop is unrolled.

3.5.2. Control Flow Loops

Arbitrary cycles in the control flow graph are used to describe the repeated execution of parts of a process. Each individual task found within the loop is automatically restarted when its direct predecessors have finished, even if the task has already completed its execution more than once. To avoid endless repetition, the user should attach the appropriate conditions to enter and exit the loop. In order to begin executing a loop, the appropriate control flow synchronization must be selected, i.e., the dependencies leading into it should be *or*'ed with the loop dependencies.

In case of loops spanning through all tasks of a process, the user should indicate which of the tasks in the loop is started first. If only one of the tasks receives data directly from the process input parameter, this task is chosen as the first task of the loop. However, in the general case, the user may have to include an additional task, external to the loop, with no incoming control flow dependencies. This task is executed once at the beginning of the process and it is linked to the first task in the loop with a control flow dependency.

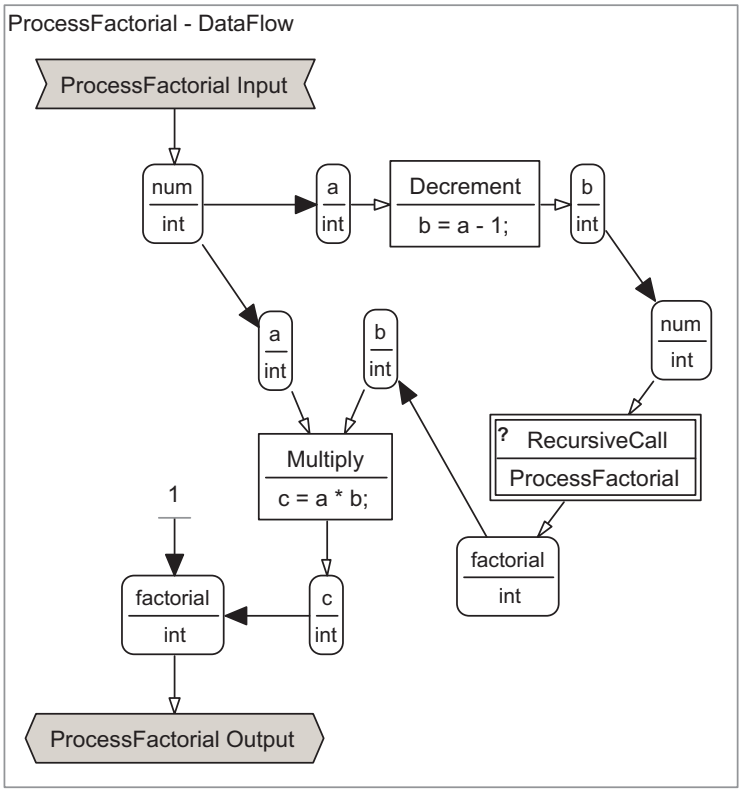


Figure 3.10.: *Data flow view of a process to compute the factorial of an integer value*

3.5.3. Recursion

Another possible way of modeling repeated behavior is through recursion. In the simplest case, this can be achieved with a subprocess referring to its container process. This way, the tasks composing the process will be repeated as long as the condition associated to the subprocess making the recursive call holds true.

As an example, Figure 3.10 shows the data flow graph of a recursive process, which computes the factorial of a number. In the example, two tasks computing Java expressions to decrement one number and multiply two numbers are linked with a subprocess which recursively calls its container process. The condition associated with the subprocess stops the recursion and forces the process to return the constant value of one.

3.6. Comments

In most programming languages, comments are very important to enter humanly readable descriptions of parts of the code. Also with the JOpera Visual Composition Language the user may attach a description to each process, as well as to each component service. This description complements an object's name by further

specifying what the process or the component is intended to do. As with most visual editors, further user comments can also be visually inserted into any of the data or control flow graphs by means of text boxes (rendered with a typical yellow post-it note flavour).

Another typical usage of comments in ordinary programming languages, is the temporary removal of program code, which is "commented out" so that it will be ignored by the compiler, while it still remains visible to the user. Given the practical importance of such way of using comments, also in the JOpera Visual Composition Language we support this approach. By visually stretching a comment box so that it overlaps with existing parts of a diagram, the user may temporarily disable the compilation of such diagram elements so that they will be ignored by the compiler and will not be part of the execution.

3.7. Reflection

In this section we present the reflection features of the JOpera Visual Composition Language. Reflection is the ability of a computational system to represent and modify information about itself [151]. In the JOpera Visual Composition Language, reflection is used to access metadata both about the static structure of the process and about its state of execution, as well as about its runtime environment.

Reflection is very important in a language intended for service composition, as it uses a well defined syntax to expose and give controlled access to the *system parameters* of the specific type of services that are composed. Furthermore, through the invocation of *system services*, it is possible to model within a process the interaction of a process with its environment. This can be used, for example, to access JOpera's directory services, in order to discover what are the available providers for a given service interface and, as we will show in Example 3.2 on page 36, to model the late binding of a service implementation to its interface. The combination of reflection with the list-based loops is an useful technique to enhance the reliability and decrease the response time of a service invocation, as we will discuss in Example 3.4 on page 39.

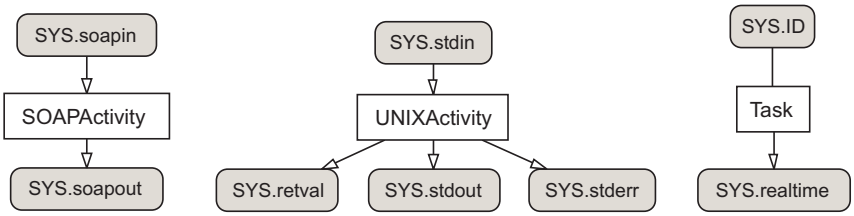


Figure 3.11.: *Example of system parameters and properties*

3.7.1. System Parameters

In addition to the data flow parameters defined by the user, each task is associated with a set of system parameters and properties which can be used for a variety of purposes. In general, they contain metadata about the execution of the process and their values are updated automatically by the runtime environment. System *input* parameters can be used to control the behavior of a task, e.g., setting its scheduling priority, and can be connected with incoming data flow bindings like any other user input parameter. System *output* parameters are used to read metadata about tasks, e.g., their running time, and are connected with outgoing data flow bindings to other system or user parameters. Similarly to user output parameters their value is – by definition – available only after a task has completed its execution. One exception to this rule are *system properties* which are also used to read metadata, but their value can be read at any time, i.e., both before and after a task is executed. Therefore, a data flow binding involving a system property does not imply a control flow dependency.

The same visual syntax applies to both system and user data flow parameters, with the only difference that the former are colored in gray and their name always begins with the `SYS` prefix. System properties are linked to their task with an undirected edge, symbolizing that their value can be read also before the corresponding task has been executed.

The set of available system parameters depends on the type of component associated with the task, and changes for processes, subprocesses or activities[4]. Figure 3.11 shows the visual syntax of system parameters and properties with some examples. In the case of activities representing Web service calls, the two system parameters called `soapin` and `soapout` give direct access to the XML content of the SOAP request and response messages (3.11.a). Similarly, for activities executing UNIX programs, the `stdin`, `stdout` and `stderr` standard data streams are provided together with the `retval` parameter, which contains the exit code of the program as it is returned by the operating system (3.11.b).

Each task is associated with a system property called `ID`, which can be used to uniquely identify the task among all other tasks of the process and among all instances of the task that have been executed by JOpera (Figure 3.11.c). This property is typically used to generate unique filenames for storing the results of the task, as it guarantees that they will not be overwritten by other instances of the same task that are running concurrently.

For execution profiling purposes, JOpera measures the execution time of each task of a process. This information is displayed to the user in JOpera's process monitoring environment. In addition, the same information can be accessed from within a process, in the form of system parameters (`cputime`, `realtime`, `walltime`) associated with each task (Figure 3.11.c).

[4]See Chapter 4 for more information on the relationship between system parameters and component types.

3.7.2. System Services

System services expose information about JOpera's runtime environment and let a process interact with it. They currently include: the program library API, the process control API and the resource management and scheduling API. As opposed to the system parameters, tasks invoking system services are not represented differently from tasks calling other types of components.

The system services of the process control API are mainly used for controlling the execution of a process from within the process itself. This enables, for example, to cancel the execution of a process upon detection of a certain condition. Similarly, it is possible to automatically suspend a running process upon reaching a certain stage of the execution and have a user manually resume it when appropriate.

Examples on using the resource reservation service and the program library system services for dynamic late binding are presented in the rest of the section. In particular, we will show how to combine late binding and reflection with the list-based loop operators to enhance the reliability of a service invocation, if multiple, alternative service providers are available.

Example 3.2: Late Binding

Reflection can be applied to a Web service composition scenario, where typically the services published on the Web have a variable degree of availability and tend to evolve quickly, especially after the processes composing them have been defined. Through late binding and the ability to gather information about the available services, a process can be made more resilient to these changes as it is dynamically adapted to the environment where it is running.

The example of Figure 3.12 illustrates how to use system parameters to support *late binding* of tasks to services. The choice of which service (or process) to invoke when executing a certain activity (or subprocess) is done dynamically based on the value of the `prog` (or `proc`) system parameter. This value is normally set at compile-time, but can also be changed at run-time, both by the user and from within a process.

More in detail, the example data flow graph shows how to use the `prog` system parameter to set the service that will be invoked by the `CallService` activity. The name of the actual service is retrieved using the `LookupService` system service, which attempts to locate a fitting service implementation given the `interface` of the activity (identified by its `ID` system property) and the additional constraint on the service's `name` provided by the `service` process input parameter.

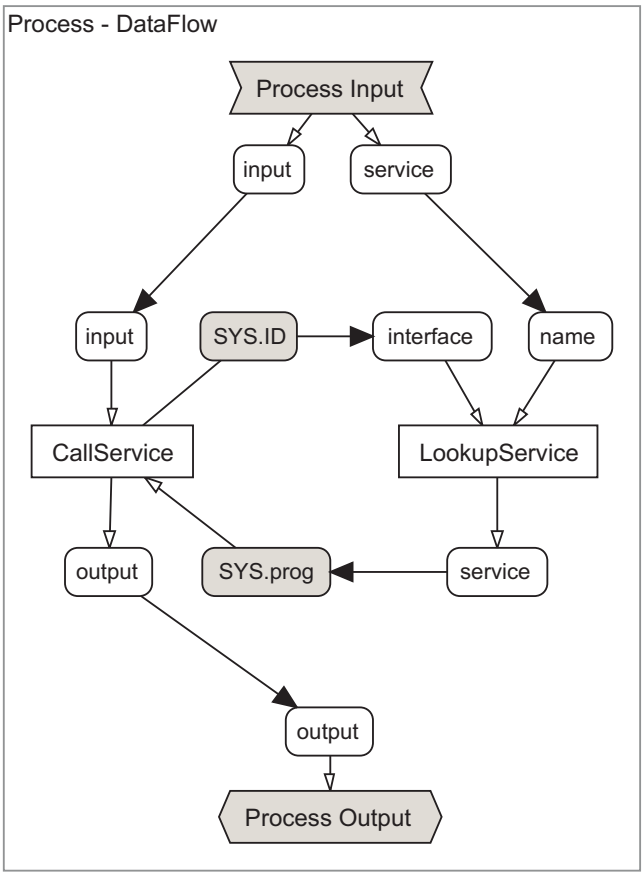


Figure 3.12.: *Data flow view of the late binding example*

Example 3.3: Cluster Resource Reservation

In a cluster computing environment, it is useful to program computations in a parametric way with respect to the available computing resources. These computations are normally developed in a small testbed, while in production settings they should scale to use a larger amount of computing power. To enable process portability, it is important to keep the process model independent of the characteristics of the environment in which should run and let the system to the necessary adaptations at runtime. To do so, the ability to inquire at runtime about the number of nodes that can be reserved to perform a parallel task can be very important, e.g., to dynamically determine the optimal partitioning of the data [4].

The example in Figure 3.13 shows how to use the `ReserveResource` system service. Behind it, there could lie the API of a complex resource management and scheduling system, which has been greatly simplified for the purposes of

this example. This system service receives one parameter called `size`, which contains the number of desired nodes and returns the identifier of the `group` of nodes that has been reserved as well as its `size`, indicating how many nodes could be reserved. The former is passed to the `resources` system parameter of the `Compute` subprocess. This parameter has the effect of constraining the execution of the content of the subprocess to the given group of resources. The `size` output parameter is then passed to the `DataPartition` task which uses it to prepare the `list` of work items to be computed.

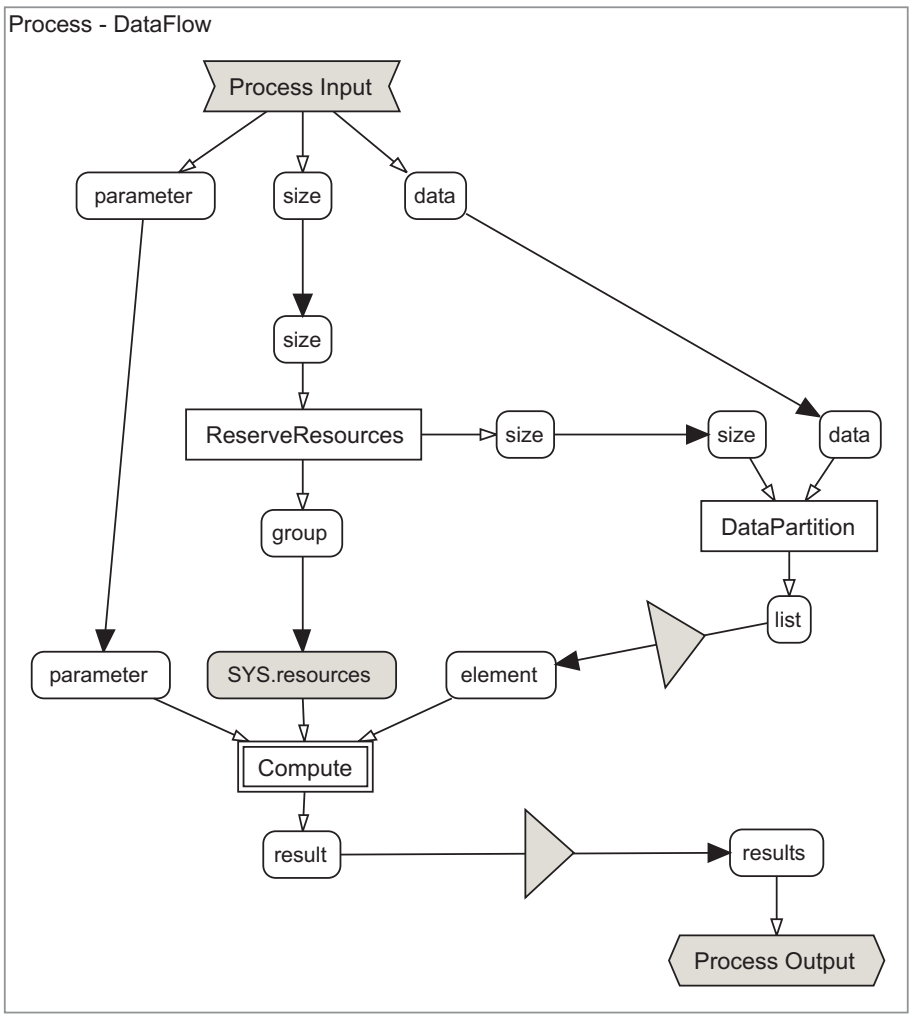


Figure 3.13.: *Data flow view of the cluster resource reservation example*

Example 3.4: Reliable Service Invocation

Building applications out of composite Web services can lead to disastrous results as soon as one of the services becomes unavailable at run-time. However, the reliability of the composite service defined with JVCL can be increased through redundancy. Assuming that a set of equivalent, alternative service providers are available, with JOpera it is possible to use reflection together with various exception handling and synchronization techniques to model the invocation of a service chosen from alternative providers and control precisely what happens in case of failures.

A Web service can become unavailable for many different reasons. The connection across the Internet to the Web service provider can fail during a SOAP message round. The Web service may have been taken offline temporarily for maintenance. The Web service might have been renamed or moved to a different server and, as a consequence, the binding information in its WSDL description may be out-of-date.

Whatever the reason, from the point of view of the composite application, a failure to contact a Web service can be very similar to dereferencing an invalid pointer in traditional applications. If no corrective action is taken, it may lead to the failure of the whole business process. To ensure a successful invocation, failure handling actions can be taken at different levels in the communication stack as well as at the process level itself.

Dealing with communication failures

To deal with problems at the communication layer, the SOAP message can be transferred with an asynchronous messaging or queuing system, instead of using a synchronous HTTP connection [267]. Although its latency may increase, the communication becomes resilient to temporary communication problems, as the messaging system can store and forward the message when an appropriate path to the service provider can be found. A similar approach can also help with temporary outages of the server itself.

If the location of a WSDL interface description is not changed, and the interface of the service is not modified, changes to the binding (for example, the server's location has been moved to a different host), should remain transparent to the process because the latest version of the WSDL can be fetched before reattempting the call. If a WSDL contains multiple bindings, the SOAP communication library may attempt to contact all of them before reporting a failure of the invocation back to the application [110].

Handling failures at the process level

All of the previous steps are usually taken inside a SOAP communication library and can be controlled by specifying the appropriate bindings in the WSDL description of the Web service. Although these mechanisms can improve reliability in case of short, temporary outages, the service invocation will still fail if the service cannot be contacted after a certain time. Therefore it is also important to deal with such failures at the application level, assuming that a suitable equivalent service implementation can be invoked at many alternative service providers. In JVCL there are a set of language constructs to deal appropriately with irrecoverable failures of a certain service invocation. This means that, when everything else fails, it is still possible to model explicitly, at a high level, what to do, using both a form of *retry on exception* and, with some restrictions, an advanced synchronization technique applied to a multi-instance pattern [233], both of which we will illustrate in the rest of this section.

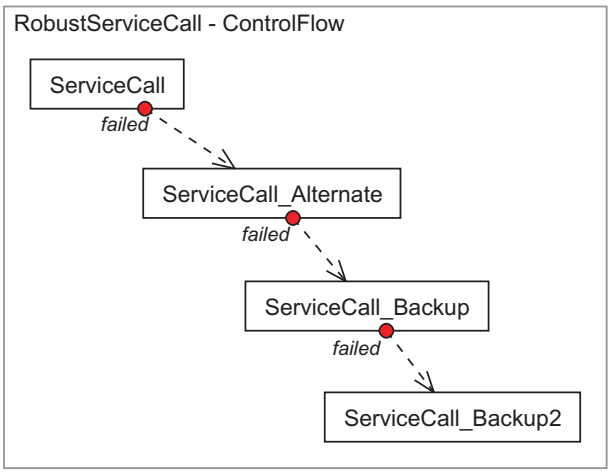


Figure 3.14.: *Sequential invocation of alternative services*
This chain of four Service invocations is traversed as each call fails. JOpera attempts to contact the four alternative, equivalent services in the order specified by the control flow graph drawn by the developer.

Exception Handling The first approach is based on the basic exception handling construct of JOpera. As shown in Figure 3.14, two service invocations can be connected by a `failed` control flow dependency, which will be triggered only if the first invocation fails, so that the second one can be tried in its place. The example shows this pattern applied to four services, which are connected in a *failure-triggered chain*. They are to be invoked sequentially, but only if the previous service in the sequence fails.

This approach has the advantage that it is rather easy to program: in

JOpera all that is required is to make a duplicate of a service invocation (including existing data flow connections), substitute the target service to be invoked with an equivalent one and connect the two invocations with the exception handling control flow dependency. However, in our experience, this is an inflexible solution, because both the set of services to be tried and the order of the attempts is fixed and cannot be changed once the process is deployed. These limitations are overcome by the next construct, which allows the usage of a dynamic set of alternative services and does not constrain the order in which they are contacted.

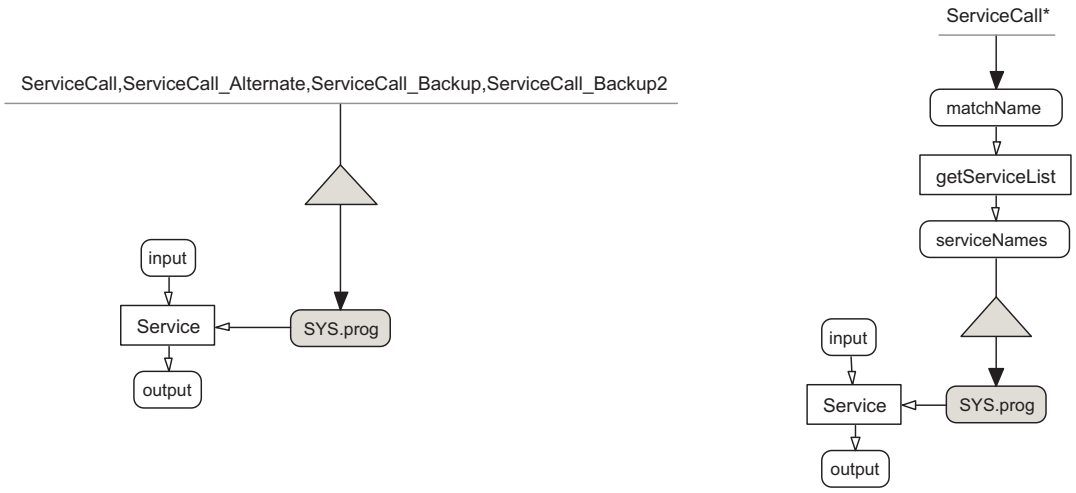


Figure 3.15.: *Parallel invocation of alternative services*
This data flow graph represents the parallel invocation of the same four services of Figure 3.14. In the static implementation (left) the list of services is hard-coded, while in the dynamic implementation (right) the list of services is retrieved at runtime. The split operator can be configured so that the invocation will complete as soon as one of the services will respond successfully (Figure 3.16).

Parallel Invocation Not only the availability of the various services may change at runtime, but also their response time may depend on the current load at the different service providers and on the network congestion. The previous model captures the cascading retries that are performed if a service in the sequence of alternative invocations doesn't respond or fails. Now, although the services are alternative and equivalent, their order in the sequence is fixed at design-time and it is always the same for all executions of the process. In some cases, it may be useful to use a different pattern. Instead of modeling this behavior as a sequence of invocations, we use a set of invocations which are all started at the same time. The result of the first one which successfully completes is taken and all others (even if they don't fail) are simply ignored (or aborted). This way, the process is again resilient to failures of all but one of

the Web service involved. Additionally, its performance is potentially better, as the call completes as soon as one Web service responds. In case of failures, the execution is not delayed by the calls that cannot be completed.

We can model this behavior in JOpera by using the *parallel split operator* applied to a list of service names (Figure 3.15). This way, the invocation of each alternative service in the list will be initiated in parallel. The synchronization condition associated with the split operator can be set to make the parallel invocation terminate as soon as one service returns without a failure (Figure 3.16). With the appropriate settings, we can achieve both optimal response time (as the result from the most responsive service is taken, while all others are ignored) and we can choose to ignore the service invocations that failed.

In Figure 3.15 (left) the list of services is still hardcoded in the process, therefore the example does not yet solve the problem of modeling a dynamic environment, where the set of Web services which can be used changes after the process which composes them has been written. To support this scenario we use another one of JOpera's reflection features, which models the runtime discovery of the available services matching a certain criteria (for example, the name or the required input/output interface). As shown in Figure 3.15 (right) the list of services to be invoked in parallel doesn't need to be hardcoded, but can be parametrized based on the results of a query to the system registry service.

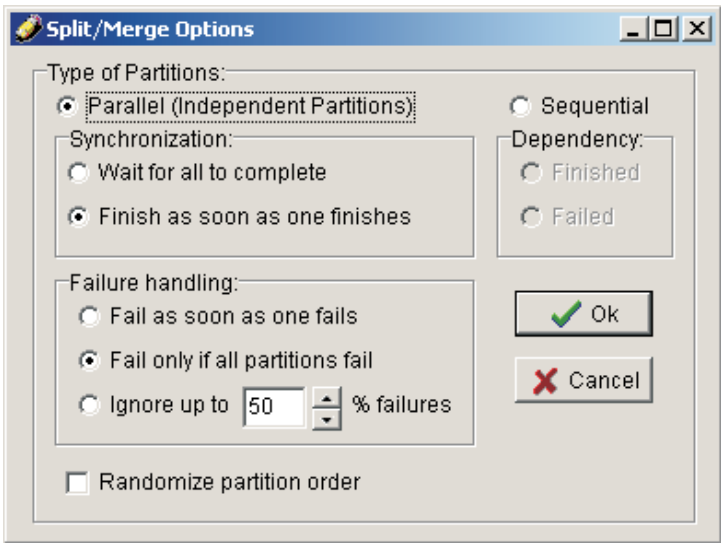


Figure 3.16.: *Split/Merge Options*
A screenshot of JOpera showing the options which control the behavior of the Split/Merge operators. With them the developer can finely control if alternative services should be invoked in parallel, or sequentially, and, in the first case, if failures in invoking some of the services should be ignored.

Sequential Invocation This solution, based on the parallel invocation of alternative services, is only applicable to stateless services we can afford to invoke in such manner. In some cases, for example when ordering a book, it may be necessary to try the various providers one at a time. To do so, we can select the *sequential* split operator (Figure 3.16) and still keep the previously described list-based patterns. The only difference being that the split operator is configured to invoke each service sequentially, and only if the previous one in the list fails.

Finally, by activating the `Randomize partition order` option, it is possible to conveniently shuffle the services in the list so that the sequence of invocation is not fixed to any particular order and the load at the various alternative providers is kept more balanced, as each execution of the process will attempt to invoke the service providers in a different order. This feature has been added for convenience only, as it is was already possible to explicitly add to the data flow a task which randomly reorders the list of services used to drive the parallel (or sequential) invocation.

3.8. Discussion

In this chapter we have presented the JOpera Visual Composition Language. This language is intended to be used as a generic glue language [82] for coordinating collections of software components, where the order of execution of services, the data exchanges between them and the necessary failure handling behavior can all be specified with a simple visual syntax. As we have shown with several examples, the language is expressive enough to be applied to realistic settings.

In particular, we found it useful to include in the JVCL the visual representation of both the data flow and the control flow graphs of a process. As we have discussed in Section 3.4, it is possible to automatically derive the control flow graph from the data flow. Like in data-driven data flow languages a task cannot be started until all of its data dependencies are satisfied [104]. Unlike in traditional data flow approaches we include an explicit description of the control flow of a process in order to provide an overview of the order of execution of the tasks. Furthermore, developers may use it to specify additional control dependencies that cannot be derived from the data flow. It should be noted that the syntax used to specify the control flow of the process has been intentionally left quite underdeveloped, as a directed graph layed out in two dimensions is already a good visual representation of the partial order of the execution of the tasks. With it, non-linear dependencies modeling parallel execution, branches and synchronization points in the control flow can be visualized in an intuitive way. Furthermore, the nodes of the graph can be annotated with conditions to model alternative paths in the order of the invocation of the services.

Nevertheless, the simple graph-based syntax of the control flow could be ex-

tended in different directions. One possibility would be to add syntactical "sugar" to model branches and synchronization points explicitly as nodes in the control flow graph. In a similar way, the conditions associated to each task could be specified visually. However, it remains open to discussion whether this would be a significant improvement with respect to the current textual approach, where a condition is entered as a boolean expression.

Another idea consists of improving the support for the visual specification of properties shared among a set of tasks. For example, *swimlanes* [30, 99] could be used to partition the diagram space in order to specify scheduling constraints associated with a group of tasks. To do so, tasks are visually assigned to the resource (or the actor, role, owner) responsible for executing them by positioning the tasks within the corresponding region of the diagram. Currently, such properties can be specified by using reflection with the assignment of the owner of a task to the appropriate system parameter.

Similarly, a transactional view over the control flow graph could be added to model ACID properties associated to subsets of tasks that should be executed atomically or in isolation from other concurrent instances [206].

Given the large amount of existing contributions in these areas (formal models [162, 190, 234, 256], transactional properties [10, 148, 206], exception handling and recovery [97], advanced control flow patterns [233]), we have chosen to use a simple control flow syntax while focusing on an explicit and richer visual representation of the structure of the service composition in terms of its data flow [62, 104], an aspect that has been overlooked by most process modeling efforts.

As a final remark, in our design of the visual language we made very few assumptions about the actual type of services to be composed, attempting to keep the visual composition language as general as possible while clearly separating compositional aspects from the model of the individual component services [82]. In a JVCL process, composition is defined at the level of service interfaces, mainly in terms of the data flow bindings between their input and output parameters. As we will present in the next chapter, this parameter based model of service interfaces can be mapped to many different types of service invocation mechanism ranging from coarse-grained Web service invocations to fine-grained scripts written in Java.

4. Component Types

In this chapter we describe JOpera's open component meta-model. As JOpera is a system for service composition, here we give some insight about what are the shared properties of the services that JOpera uses as components.

First of all, it should be emphasized that JOpera provides extensive support for many different types of components with the goal of providing the developer with a flexible, convenient and extensible model without sacrificing efficiency (in terms of low overhead) during service invocation.

It should also be noted that introducing a component model supporting a wide variety of component types can be very useful in keeping the composition language simple.

4.1. Motivation

In some of the existing process based systems for service composition (e.g. [29, 112]), the services to be composed are all assumed to be of a single type: Web services. It should be noted that when facing software integration problems at an Internet-wide scale, Web services seem to be the most appropriate solution [89]. However, for other kinds of deployment settings and service integration scenarios, other types of components are still likely to be used. More specifically, in the context of the JOpera Visual Composition Language presented in the previous chapter, it would be an unnecessary restriction to assume that JOpera's component services must all be Web service compliant. In fact, there are many existing, well established service access protocols that should not necessarily be considered as out of date, when compared to Web services [3]. As stated by the jbpm.org project [18]:

> BPEL4WS, BPML, WSCI are all "workflow standards" based on web services. While web-services are cool and a nice buzzword, we think it is a big limitation to restrict a workflow engine to only Web services. There are so many other nice protocols like HTTP, RMI, CORBA, EJB, TCP/IP, UDP/IP, JMS, ... As a workflow engine is mostly used for enterprise application integration, it seems ridiculous for an engine to support only Web services and ignore all other protocols. In our opinion, a workflow engine should communicate with each system in the technology that is most appropriate and not force the development and maintenance of Web service wrappers.

We fully agree with such a view, and have also designed the JOpera system to support components than can be accessed through a variety of protocols, including, but not limited to Web service compliant ones. This way, the developer of a composite application is not limited to use components accessible only through the SOAP protocol and, in case of components supporting more protocols, the system can use the most appropriate one in terms of performance, security and reliability.

The need for supporting a variety of service access protocols is also recognized in the Web services community. To this end, the WSDL interface description standard supports an open-ended set of transport protocols. Therefore, a Web service, whose interface must be described using WSDL, does not necessarily need to be invoked using the relatively slow SOAP protocol if the client understands other (non standard) protocols which may offer better performance. Currently, however, alternative protocols are not yet widely supported and as long as they are not standardized, using them would defeat the main point of the Web service vision, where everything should be standardized in order to achieve widespread interoperability [5].

Moreover, even when following this approach, in order to bridge the gap between the existing component heterogeneity and the uniform Web services standards, wrappers and interface adapters are still required, to make the "legacy" types of components and protocols fit with the new standards. This approach both introduces additional, unnecessary execution overhead and shifts development and maintenance costs from the infrastructure to the end user [174].

As we will present in this chapter, in JOpera we have chosen not to restrict the types of component services to Web service compliant ones, described by a WSDL document and accessible through SOAP (Section 4.3). Additionally, a JOpera component can represent, for example, the execution of a UNIX or Windows command line in the operating system shell (Section 4.4), a remote procedure call or method invocation (Section 4.5), a job submitted to a batch scheduling system of a cluster of computers (Section 4.9), an SQL query to be sent to a database (Section 4.6.2), and an XSL style sheet transformation to be applied to some XML data packet (Section 4.7.3). Furthermore, for modeling services built out of fine-grained operations, small scripts written in Java can be directly and efficiently embedded in a process (Section 4.5.1). From these examples it can be seen that, with JOpera, the developer may conveniently choose the most appropriate component type in terms of the effort required to integrate it into a process and still be relatively sure that the runtime overhead of accessing the service will be as small as possible.

Finally, as it would be impossible to provide out-of-the-box support for all possible kinds of components, we will also discuss how to extend JOpera's open component model. Considering the previous discussion, we believe it is less expensive to build *once* a generic adapter to integrate a certain type of components into JOpera, instead of having to setup a different Web service wrapper for each of the components of that particular type that have to be called from within a process.

4.2. Component Meta-Model

Before describing in detail the properties of each of the component types currently supported by JOpera, we introduce JOpera's component meta-model. This way, we both motivate the flexibility and the extensibility of the component model and summarize the information required to model and to access each type of component.

JOpera's model of a component type mainly defines a set of attributes describing how to invoke the component service's functionality and how to structure the data exchanged with it. More in detail, when adding a new component type to JOpera's model it is necessary to define and design:

1. What is the set of *system parameters*. Depending on the specific type of component, a different set of system parameters may be used to control the service invocation and to access related metadata. The values of these system input parameters are set at design time, either when registering a new service with the system's component library or when composing the services with other ones. In general, most of the system input parameters can contain placeholders that are substituted with the values of the corresponding user-defined input parameters at runtime. Finally, the system output parameters store the raw result of the service invocation.

2. How to map *control flow events*. Basic control flow events include: the starting of the service invocation and the corresponding notification that the service invocation has completed. More sophisticated events may involve the interaction with an ongoing service invocation, i.e., the ability to abort, suspend and resume it.

3. How to *schedule* the invocation. A service can be invoked either synchronously or asynchronously, with respect to JOpera's internal threading model. A synchronous invocation involves less overhead but can delay other concurrent invocations. On the other hand, asynchronous invocations are queued[1]. Furthermore, for asynchronous component types, JOpera may choose among a set of alternative providers where the service should be invoked. For other component types, this form of scheduling may not be an option, due to performance or protocol restrictions.

4. How to map *data flow*. Input and output data needs to be transferred back and forth between JOpera's parameter-based representation and the service's own representation.

5. How to interpret *failures*. Not only do service invocations finish; sometimes they fail. Depending on the type of component, failure detection may be based on different assumptions.

[1]See chapter 7 for more information on the various service invocation mechanisms.

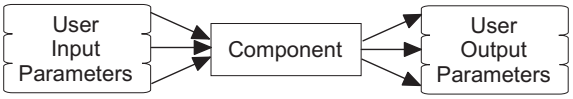


Figure 4.1.: *Data flow interface of a component*

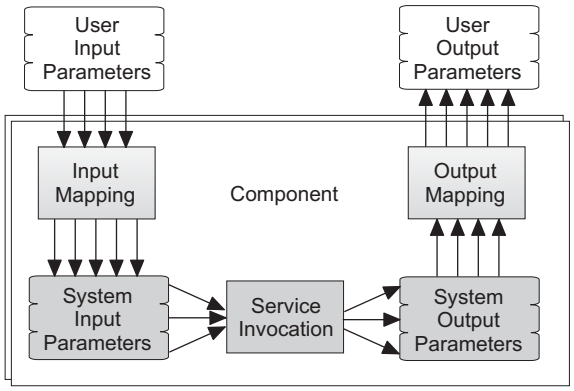


Figure 4.2.: *Data flow mapping inside a component*

4.2.1. Data Flow Mapping

From the point of view of transferring control, the invocation of a large number of different component types is not so difficult to model, as this amounts to describing the invocation of the service and the corresponding notification that the service's invocation has completed [209].

In our experience, however, a more difficult challenge lies in modeling the data to be exchanged with the service and in how to map JOpera's parameter based representation (Figure 4.1) to the service's internal one. For some component types this can be relatively simple, at least from a syntactical perspective, where standards, e.g., SOAP, define how to format the input data and how to interpret the output data. In other cases, e.g., when integrating legacy UNIX applications, the problem is much more difficult and there is no general solution, i.e., the ad-hoc development of wrappers may be required.

In order to provide the necessary flexibility to integrate several different component types, in JOpera we follow a two step approach to address the problem of mapping user-level data parameters to the actual structure of the data understood by the component type. At this point, it should be noted that user parameters are application dependent and therefore have nothing to do with the system parameters, which instead model the information required to access a particular type of service. The mapping between user (application) parameters and system (component type) parameters is specified once, when a new service component is registered with JOpera. This mapping can be derived automatically, e.g., by reading the WSDL description of a Web service.

The data flow mapping depicted in Figure 4.2 can be formally represented as a

composition of two mappings (m_i, m_o) which are applied to fit the input and output parameters of a certain service call C to the given interface S. More precisely, the interface of a service contains a set of user-defined input ($[I]$) and output ($[O]$) parameters:

$$[O] = S([I])$$

Furthermore, a set of predefined service types C_t are available. These define the representation of the corresponding access mechanisms and invocation protocols in terms of input ($[i]$) and output ($[o]$) system parameters:

$$[o] = C_t([i])$$

In order to bind a service interface to an implementation of a given service type, it is necessary to provide the corresponding input $m_i : [i] = m_i([I])$ and output $m_o : [O] = m_o([o])$ mappings. At runtime, these mappings are composed with the invocation of service of a given type as follows:

$$[O] = S([i]) = m_o(C_t(m_i(I))$$

Following such mapping, before a service can be invoked at runtime, the user input parameters are translated to its system input parameters. The main mechanism to model and perform this mapping (m_i) consists of using parameter placeholders, which identify one user input parameter and are replaced with its content when the mapping is evaluated. These placeholders follow the simple convention of including the name of a parameter between `%` characters [98, 144].

The service is then invoked and the results are placed in the system output parameters corresponding to its type. The reverse mapping m_o from the system output parameters to the user-defined output parameters is applied. As opposed to the input mapping, where a relatively large number of user parameters are assigned to a small number of system parameters, in this case it is more complex to take the content of a few parameters, e.g., the output of a program or a Web page, and model how to extract the application dependent information. For data having a relatively well defined syntax, e.g., XML, it is possible to follow the convention of encoding parameter names as tags and insert their values between those tags [**?**]. In general, *ad-hoc wrappers* can be plugged into JOpera with the purpose of scraping the values of the output parameters from the arbitrarily formatted data produced by the service.

4.2.2. Abstract Service Types

In order to emphasize the feasibility of the approach, in the rest of the chapter we show how to apply the component meta-model proposed in this dissertation to a large number of different component types. In some cases, there should be no difficulty in recognizing the difference between the protocols and the mechanisms required to access different types, e.g., Web services (Section 4.3) and UNIX command lines (Section 4.4). In other cases, although the underlying mechanism is

fairly similar – e.g., UNIX command lines and Java virtual machines (Section 4.5.4) – we made a distinction to show that the component model doesn't have to strictly follow a classification of service access mechanisms. Thus, we also show that it is flexible enough to accommodate user-oriented distinctions, which facilitate the of use of components of a certain type.

To summarize the current set of component types supported by JOpera's component model, we have listed their most important features regarding the mapping of control, scheduling, input and output data and failures in Table 4.1. Furthermore, to give an overview over the content of the chapter, in Figure 4.3 we also present the inheritance relationships between some of the various types, as far as the definition of the set of system parameters associated to a given component type is concerned.

As it can be seen in Figure 4.3, all component types inherit these common system parameters, which will be omitted from the following descriptions.

Input System Parameters	`timeout`	The timeout system input parameter controls the maximum allowed execution time. If set, a service invocation will be automatically interrupted and failed if it does not complete within the given time.
	`wrapper`	This system parameter indicates the policy to be used in order to extract the user-defined output parameters from the system parameters returned by the component type. This parameter provides the developer with the flexibility of defining custom mapping policies in order to override the default XML-based one.
Output	`state`	The state output system parameter is set to either Finished or Failed to indicate the outcome of the service invocation.
	`realtime`	The realtime parameter measures the duration of the service invocation in milliseconds.

4.3. Web Services

These components represent the invocation of a remote service published on the Web. Currently, such services can be accessed in two ways:

1. (`SOAP`) The service's interface is described in WSDL, and the service is accessible through the SOAP protocol.

2. (`HTTP`) The service is accessible through HTTP only, and the retrieved information is formatted using HTML.

4.3.1. SOAP

This first component type models the latest form of standard compliant Web services, whose interface and location are described in a WSDL document [246]. Furthermore, these components are remotely accessible through the SOAP protocol [245]. Thanks to these standards, it is possible to automatically import the

Component Type		Control[a]	Input and Output Data		Failure
Web components					
Web Service	(SOAP)	Synch/S	SOAP	SOAP	SOAP Fault
Web Server	(HTTP)	Synch/S	CGI/URL	HTML	HTTP Error
Local components					
Shell Command	(UNIX/NT)	Synch	CmdLine, Stdin	Stdout	ExitCode, StdError
Java components					
Java Program	(JVM)	Synch	CmdLine, Stdin	Stdout	ExitCode, StdError
Java Script	(JS)	Immediate	Local Variables		Exception
Java Method	(JAVA)	Synch	Method Parameters		Exception
Java Remote Method	(RMI)	Synch/S	Method Parameters		Exception
Script components					
Script	(SCRIPT)	Synch	CmdLine, Stdin	Stdout	ExitCode, StdError
Database Query	(SQL)	Synch/S	Parameters	interesting problem	JDBC Error
XML components					
X-Path Query	(XPATH)	Synch	XML	XML	X-Path Processor Error
Style Sheet Transformation	(XSL)	Synch	Parameters	XML	XSLT Processor Error
System components					
JOpera Echo	(ECHO)	Synch	XML	XML	XML Parser Error
JOpera Process	(OPERA)	Asynch/S	Implicit Parameters and Failures		
JOpera Reflection	(JOP)	Immediate	Parameters	XML	JOpera Error
Cluster computing components					
BioOpera	(PEC)	Asynch/S	CmdLine	Stdout	ExitCode, StdError
Portable Batch System	(PBS)	Asynch	CmdLine	Stdout	N/A
Messaging components					
eMail	(EMAIL)	Asynch/S	Text	Text	eMail Server Error
Java Message Std.	(JMS)	Asynch/S	String	String	JMS Error
Business process modeling components					
BPEL activity	(BPEL)	Synch	Parameters	None	Throw
Workflow task	(WF)	Asynch/S	Text	Text	User Error

[a] As discussed in Section 7.4.2 on page 167, the invocation of the service can happen according to different control flow patterns: Immediate (Figure 7.9); Synchronous (Figure 7.10); Asynchronous (Figure 7.11); Synchronous/Scheduled (Figure 7.12); Asynchronous/Scheduled (Figure 7.13)

Table 4.1.: *Component Types Summary*

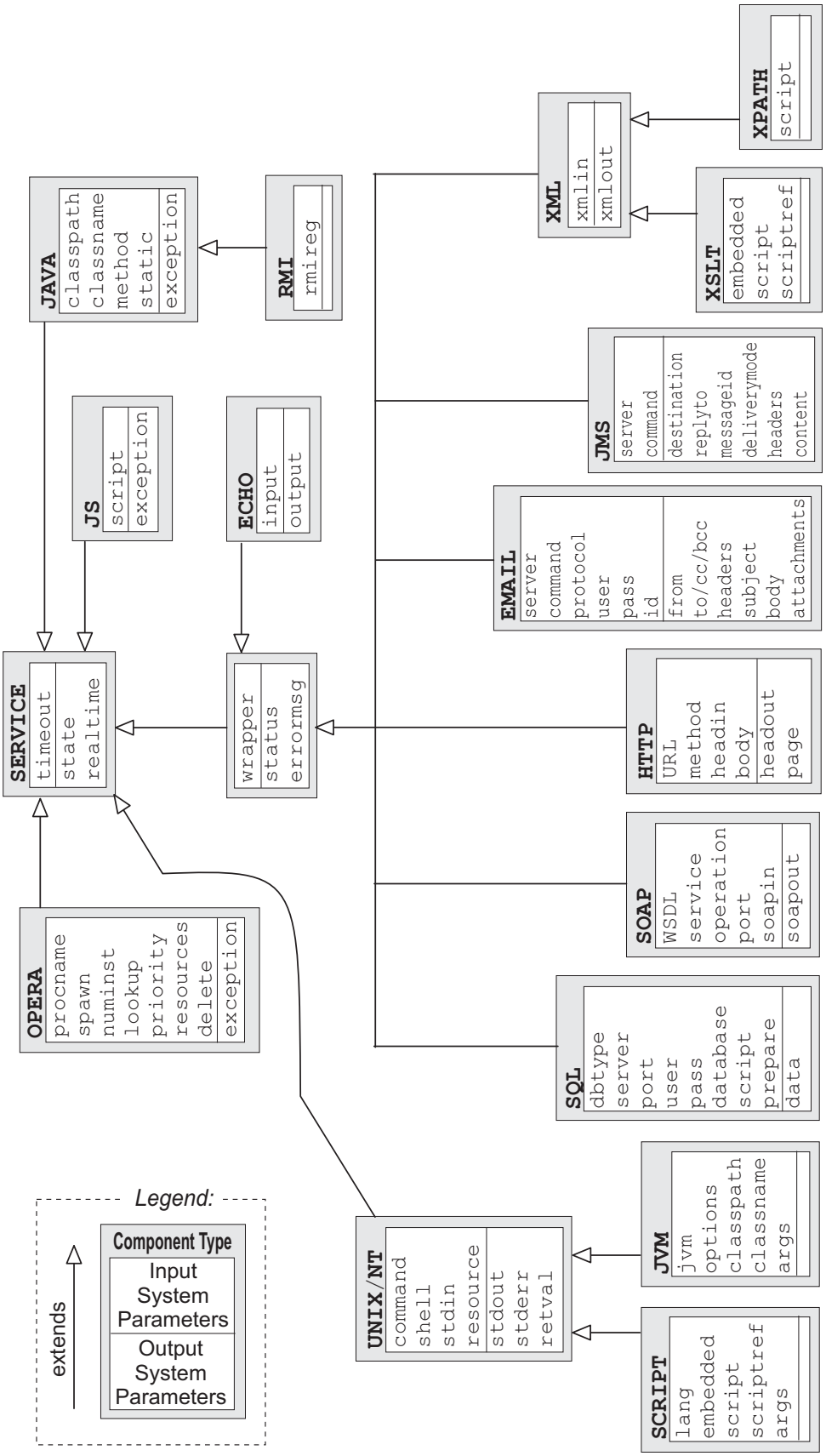


Figure 4.3.: *Summary of the System Parameters of some the component types modeled in this chapter*

service's WSDL description into JOpera's component library and use it to generate the corresponding component declarations, including the appropriate skeleton of the SOAP request messages.

Web services offer the benefit of standard-based interoperability between heterogeneous programming languages and platforms. With this technology, the effort of building systems composed out of services distributed across the Internet is greatly reduced, at the price of a relative high runtime overhead due to the nature of the protocols involved.

In the following examples of Figure 4.6 on page 57 and Figure 4.8 on page 69 we show how different Web services can be composed into a process together with other kinds of services.

System Parameters A Web service component is described by the following attributes:

Input System Parameters	`WSDL`	This system input parameter contains the URL used to locate the description of the service. The referred WSDL document contains information about the service's interface and its bindings to one or more providers and transport protocols.
	`service`	the name of the service, selected from the ones described in the WSDL document.
	`operation`	the name of the actual operation to be invoked.
	`port`	in case an operation is bound to multiple transport protocols, the value of this parameter identifies the one to choose.
	`soapin`	the body of the SOAP request message to be sent when invoking the service.
Output	`soapout`	This system output parameter contains the SOAP response (or fault) message as it is returned by the service.
	`status`	A status code which indicates whether an error occurred.
	`errormsg`	A description of the error, with debugging information.

Control and Scheduling The invocation of the service happens synchronously. In case multiple bindings are defined in the service WSDL description, the choice of the most optimal binding (and port) constitutes a form of scheduling.

Data The values of the user-provided input parameters are inserted in the SOAP request message using the previously described placeholder mechanism. In most cases, each input parameter corresponds to a SOAP message block. If necessary, JOpera escapes the content of the parameters so that it conforms to the required SOAP/XML encoding. The output parameters are filled by reading the SOAP response (if any).

Failures The invocation of a Web service may fail for one of the following reasons:

WSDL not found	The URL of the WSDL description of the service could not be dereferenced.
invalid WSDL	The WSDL description was found, but could not be understood.
invalid SOAP request	The SOAP message could not be sent because it contains invalid data.
service not responding	No response message from the service has been received after a certain timeout has expired.
SOAP fault	The service has responded with a soap fault message.

4.3.2. HTTP

In addition to standard compliant Web services, with JOpera it is also possible to conveniently retrieve information from traditional Web-based services which are accessible with HTTP only and typically format their content using HTML. Considering that there are many existing browser-based information sources and computational services available on the Web (see [19, 70, 87, 129, 139, 210] for some examples related to Bioinformatics), it becomes important to streamline the integration into a JOpera process of such type of services. It should be noted that such integration could still be feasible through other means, i.e., by invoking external HTTP client programs or by creating a WSDL/SOAP wrapper for each website involved. However, this approach would require more setup work by the user when building the process and potentially entail a higher execution overhead at runtime.

System parameters The interaction with a Web server is controlled by the following input and output system parameters.

Input	`URL`	the URL identifying the remote resource to access
	`method`	the HTTP Method (POST/GET/PUT/HEAD) to employ
	`headin`	the optional HTTP headers sent along with the request
	`body`	the optional body of the POST request message to be sent
Output	`status`	the HTTP status code
	`errormsg`	the description of the error, if any.
	`headout`	the HTTP headers of the response message.
	`page`	the content of the response message.

Control and Scheduling The Web server is contacted synchronously. By extending this model with a more precise description of the `URL` input parameter, it would be possible to introduce a form of client-side scheduling of HTTP requests among a set of alternative mirrors of a certain Web site. This way, the address of the selected mirror would be substituted into the `URL` before the request to the Web server is sent.

Data The user input parameters are encoded in the `URL` in case of a GET request or in the `body` for POST requests. The resulting HTML `page` needs to be scraped for filling the user's output parameters with their content.

Failures The retrieval of a Web page can fail for several reasons:

unknown server	The server address in the URL could not be resolved.
server not responding	The Web server did not respond after a timeout expired.
HTTP error	The HTTP `status` parameter is different than 200 (OK).

Example 4.1: Stock Quote Currency Conversion

In this example we present a process used to retrieve quotes in the desired currency for a user-provided stock symbol. This process combines two Web services, one quoting stock prices [262] and the other one performing currency conversions [261]. Although it is a simple example, it shows an application of the basic features of the language in the context of Web service composition without too many unnecessary, application related details. Furthermore, in this example we compare two different versions of the process, one emphasizing reusability, the other performance achieved through parallelism.

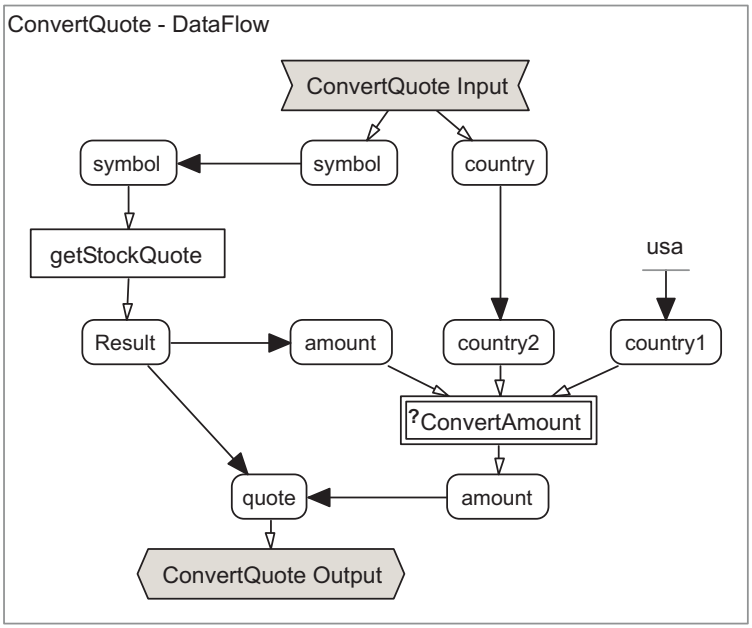


Figure 4.4.: *Data flow view of the* `ConvertQuote` *process*

The first version is the `ConvertQuote` process shown in Figure 4.4. This process takes a stock `symbol` and a `country` as input parameters and returns a `quote` for the given stock market `symbol` converted to the currency of the given `country`. The `symbol` parameter is passed to the `getStockQuote` lookup

service which returns the current price in its `Result` output parameter. This value is then passed to the `amount` input parameter of the `ConvertAmount` sub-process together with the two countries between which the value should be converted. The `country1` parameter is set to the `usa` constant value, as the price returned by the `getStockQuote` service is in U.S. dollars. The `country2` parameter is bound to the `country` process input parameter, and can be chosen by the user when starting the process. The `amount` output parameter, result of the `ConvertAmount` sub-process is copied to the `quote` output parameter of the main process.

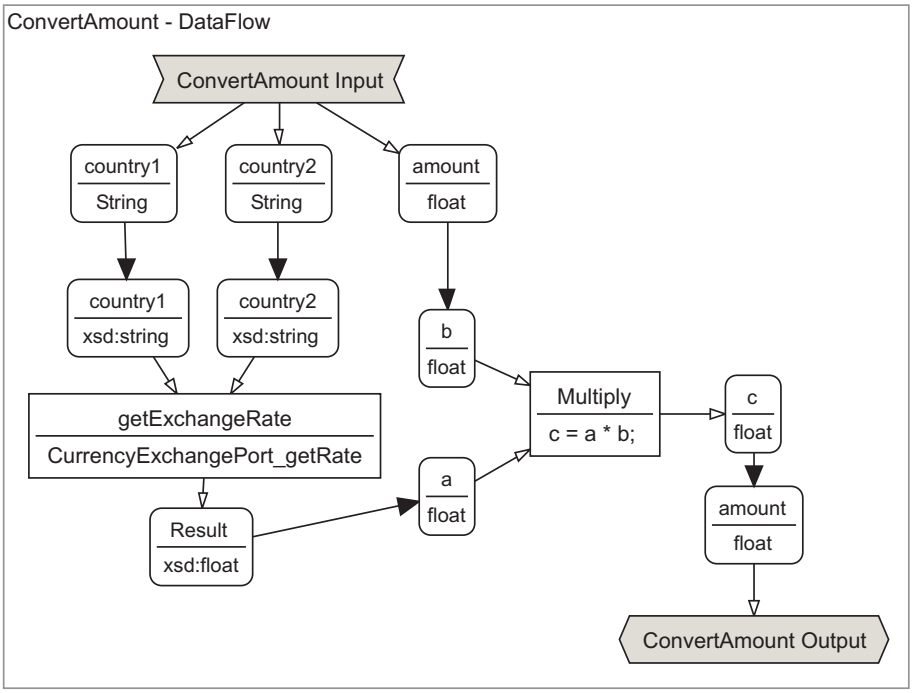


Figure 4.5.: *Data flow view of the* `ConvertAmount` *process*

The example also contains an optimization. Considering that there is no need to perform a currency conversion between identical currencies, a condition can be attached to the `ConvertAmount` sub-process to skip its execution if its `country1` and `country2` input parameters contain the same value. In this case the value of the `quote` output parameter of the process is taken directly from the `Result` of the `getStockQuote` service.

The `ConvertAmount` sub-process calls the `ConvertAmount` process (Figure 4.5), which uses a currency exchange rate service (`getExchangeRate`) and adapts its interface to perform the conversion of a given amount of currency. To do so, two input parameters `country1` and `country2` are passed to the `getExchangeRate` service, which returns the corresponding exchange rate in its `Result` output parameter. This value is multiplied with the `amount` process input parameter to compute the converted `amount` process output parameter.

This process composes services of different granularity: the slow, coarse grained invocation of a Web service (`getExchangeRate`) with the fine grained `Multiply` task, which references a Java expression used to multiply two floating point numbers. In this example, the Currency Exchange Rate service has been wrapped inside a sub-process to emphasize the reusability of this interface adaptation, which can be called from many processes.

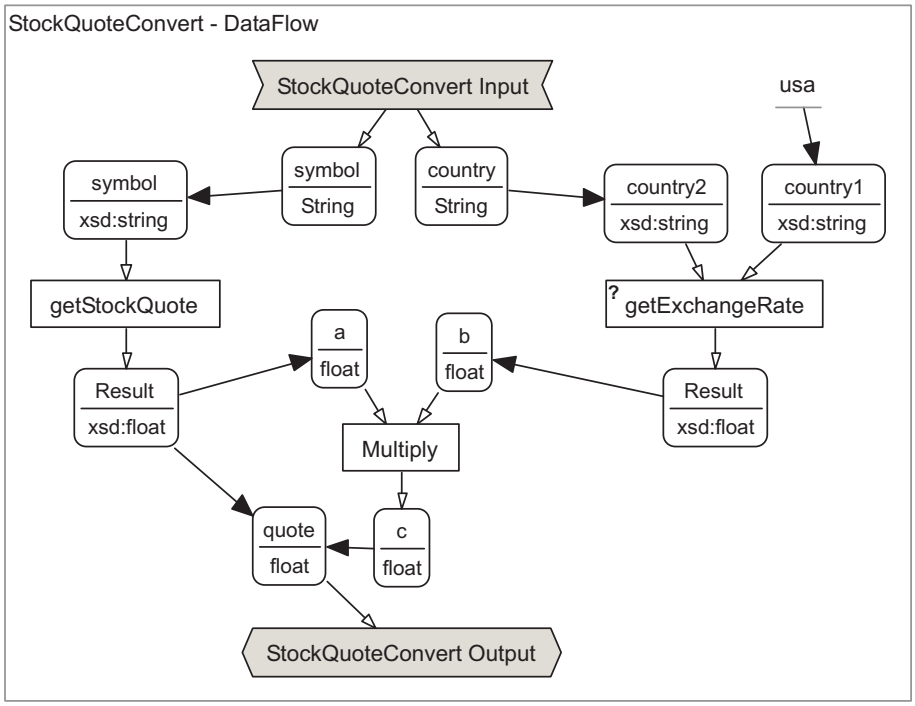


Figure 4.6.: *Data flow view of the* `StockQuoteConvert` *process*

An alternative version of the same process `ConvertStockQuote` is shown in Figure 4.6. Here all of the previously separated tasks are located within the same process. This implementation can be automatically produced from the previous example by letting JOpera expand the content of the sub-process `ConvertAmount` inside the caller process.

We have included this additional example to show that by reducing the modularity of the process it is possible to exploit the parallelism between the `getStockQuote` lookup service and the `getExchangeRate` service. In the previous example they had to be invoked sequentially, as the latter was started only after the invocation of the sub-process. In this example they are invoked in parallel, as there are no data flow dependencies between them. The retrieved `Result` parameters are merged through the previously described `Multiply` Java expression to compute the converted stock price. The fact that this process doesn't contain any sub-process invocation also contributes to a reduction in the runtime overhead, as the sub-process call is no longer necessary.

4.4. Shell Commands

Another type of components, quite different from remote Web services, are commands to be executed on a shell of the local operating system (`UNIX`, `NT`). A Shell command is typically used to provide a generic mechanism of integrating entire applications into a process. As long as these applications do not provide an explicit API, the command line may be the only viable mechanism to allow JOpera to interact with such legacy applications and control their execution. Historically, shell commands connected in a pipe and filter architectural style have been one of the earliest successful form of reusable components, where complex systems can be built out of simple combinations of subsystems [125]. Analogous to UNIX pipelines, JOpera processes can be built by drawing data flow connections between individual command invocations, with the option of specifying non linear topologies and the possibility of executing the various pipeline stages on different machines.

In other words, the services provided by essentially any executable program, which can be started by typing a command line at the prompt of the operating system shell, can be accessed with this component type. We distinguish between `UNIX` (Linux, Solaris, MacOS X, etc.) and `NT` (Windows) components to let the user enter the shell command using the syntax appropriate for the environment where the program will run, and also for catching errors, as it is not possible to run Windows applications on UNIX (and in most cases, viceversa). In order to exchange data with the external program, JOpera employs both the command line itself and pipe-based interprocess communication mechanisms.

As we will show with the Java Program (Section 4.5.4) and Scripts (Section 4.6.1) component types, the Shell Command component type can be extended to provide the user with a more convenient, template-based model of the command line.

System Parameters To execute a shell command the following system parameters are available:

Input	`command`	The command line to be sent to the operating system shell. It contains both the path to the executable program as well as its command line parameters.
	`shell`	The optional choice of the shell to use for interpreting the command.
	`stdin`	This parameter contains the data to be sent to the running program on its standard input.
Output	`stdout`	This output system parameter stores a copy of the output of the program.
	`stderr`	This parameter buffers the error messages produced by the program.
	`retval`	This parameter contains the operating system exit code. Following the UNIX convention, a value of 0 indicates a successful execution, any other value is interpreted as an error.

Control and Scheduling The shell commands are invoked synchronously.

Data The values of the user input parameters are transferred to the external program both using its `command` line and can also be copied onto its `stdin` system input parameter. By default, the standard output produced by the program is parsed following an XML syntax in order to extract the values of the output parameters, although a user-provided plugin for parsing the application-dependent output can override this behavior.

Failures The invocation of a shell command can fail for several reasons. JOpera interprets the value of the `retval` system parameter, which contains the exit code of the process as it is returned by the operating system, to distinguish between a successful execution (0) and a failed execution (non-0). In both cases, it also stores the program's standard error into the `stderr` parameter so that the user can gather useful debugging information.

4.5. Java

As JOpera's runtime kernel is written in Java, this offers the interesting opportunity to integrate various flavours of Java components into a process with different degrees of granularity and overhead. JOpera can embed small Java scripts (`JS`) directly into a process, or it can efficiently call local (`JAVA`) or remote (`RMI`) methods of Java classes and also offers a convenient way to start external Java virtual machines (`JVM`).

In an enterprise application integration scenario, where most of the business logic has been developed with Enterprise Java Beans and related technologies, it is possible to access such distributed software components from a process by modeling and invoking them as Web services. However, at runtime this would impose an excessive overhead. Especially if the process runs locally, within the same environment where most of the Java components have been deployed, the Web service interoperability and firewall-tunneling properties would not be of advantage. Furthermore, at design time, all of the beans to be integrated still need to be manually published as Web services. Although there is a growing set of tools to provide automatic support for this kind of operations, they still entail an additional development and maintenance cost, which would be reduced if the services provided by the beans become directly accessible from a process.

At the other end of the granularity scale, the Java programming language also provides a convenient way to program very small computations to be called at a certain point during the execution of a process. This way, parameter values can be easily converted, or checked for correctness. Furthermore, multiple output values can be quickly computed starting from a set of input parameters, minimizing the overhead due to parameter passing. To perform similar computations remotely using a Web service would be extremely inefficient [27]. Similarly, the values on which conditions depend on (e.g., a loop counter) can be conveniently updated with a small Java snippet, or *script*, as we will present in the following section.

4.5.1. Java Scripts

This component type models the most efficient way of embedding Java code into a process, where a small script written in Java can be executed with minimal overhead. As suggested by the previous examples (Figure 3.10 on page 33, Figure 4.5 on page 56), it can be very beneficial to use this kind of component to perform small computations in a process. If the same computation would have to be invoked using a different mechanism (e.g., Web services), the overhead of the protocols involved would make it impractical to do so[2].

System Parameters For this component type, there is only one system input parameter which contains the script itself.

`script`	The script (or Java method body) to be embedded into the process.
`exception`	If an error occurs, this system output parameter contains the message of the Java exception.

Control and Scheduling By design, the script is invoked *immediately*, i.e., within the same thread that executes a process. This has a very small overhead. However, if the script runs for too long it may delay the execution of other processes.

Data flow There is a one to one correspondence between user defined parameters and the Java variables that can be implicitly used in the script. JOpera's compiler automatically declares Java variables for each input and output parameters. For this reason, and for this component type only, it is not allowed to have input *and* output parameters with the same name, as they would be mapped to the same Java variable. After the script has completed, the values assigned to the Java variables are copied into the corresponding output parameters.

Failures JOpera detects a failure if a Java exception is raised and it is not caught during the execution of the script.

4.5.2. Local Java Method Calls

For more complex Java code that cannot be embedded into a process as a script like in the previous case, JOpera provides the local method call component type. More precisely, it is possible to invoke directly any static methods of a given class. For other, non-static, methods, JOpera first creates an object of the class (assuming that the class supports the empty constructor) and then calls the specified method of the newly created object.

[2]This issue has also very recently surfaced within the BPEL4WS [112] community, where a proposal called BPELJ is currently under discussion. Very briefly, it suggests to extend this Web service composition language with a new "keyword" (or activity type) to incorporate so-called Java snippets into a process [111].

System Parameters The following system parameters are used to identify the method and the class to be invoked:

`classpath`	The Java class path where to find the class to be loaded.
`classname`	The fully qualified name of the class.
`method`	The method to invoke.
`exception`	If an error occurs, this system output parameter contains the message of the Java exception.

Control and Scheduling The dynamic loading of the class and the method invocation happen synchronously within the same Java virtual machine which runs the JOpera kernel.

Data flow JOpera's input parameters correspond directly to the method's parameters. There is only one output parameter which contains the value returned by the method.

Failures Any uncaught exception raised during the method invocation will fail this component type.

4.5.3. Remote Method Invocations

System Parameters In addition to the parameters of a local method invocation, a remote method invocation requires the following system input parameter:

`rmireg`	The address of the RMI registry to use when looking up the name of the class to invoke.

The mappings for Control, Data and Failures are equivalent to the previously described Local Method Invocation component type (Section 4.5.2).

4.5.4. External Java Programs

For convenience, it is possible to model external Java programs as a different component type, altough the execution semantics are very similar to the Shell Command component type. In fact, this is an example on how to conveniently extend the Shell Command component type with a predefined command line. As opposed to the previous Java related components, where parameters are passed on the stack of a Java method call, in this case data is exchanged with the external JVM through the same (and more expensive) mechanisms used with the Shell Command component type.

System Parameters To call an external Java program it is necessary to specify the following system parameters, which are used to build the command line for inoking an external JVM

Input	`jvm`	The optional choice of the Java virtual machine to use.
	`options`	The JVM options to use, if any.
	`classpath`	The Class Path where the external JVM searches for the class to be loaded.
	`classname`	The name of the class to execute.
	`args`	The command line arguments passed to the main method of the class.

The mappings for Control, Scheduling, Data, and Failures are equivalent to the previously described Shell Command component type (Section 4.4).

4.6. Script Components

Scripts are components that involve the execution of an external program (or script) written in traditional scripting languages, including but not limited to PERL [250], Python [240], or domain-specific scripting languages, such as Darwin [86]. In this category we also include database scripts written in SQL [38].

Scripting languages have been traditionally a very successful form of programming *glue* code, as they enable developers to automate the interaction between a set of applications, which are reused as coarse grained components [171, 203]. We have included scripts in the JOpera component model for the following reasons.

First of all, it frequently occurs that the interaction between different applications is initially automated using this kind of technology [182]. However, following this bottom up approach, after such scripts reach a certain level of complexity, or if there is a need of integrating scripts written in different languages and intended to be run on different platforms, current scripting environments do not offer a viable solution. By making it easy to invoke external scripts from within a process, it becomes possible to build a process (or a visual meta-script), which defines how the various scripts interact.

Furthermore, scripts are great for developing *wrappers*, where the interface of "legacy" applications can be non invasively modified to fit with the rest of the process. Once such wrapper is available, it should be as easy as possible to invoke it from a process.

Finally, thanks to JOpera monitoring environment, it becomes possible to track the progress of such processes built out of scripts, a task which usually requires additional programming effort with traditional scripting languages.

4.6.1. Scripts

In order to execute a script, JOpera invokes the scripting language interpreter and passes it a) the script and b) the input parameters. Similar to invoking an external Java program, this component type is an extension of the Shell Command component type. The script to be executed can be stored into an external file or can be also embedded into the component description. In practice, in order to enhance the portability of the service definitions, it can be quite useful to store such scripts, especially if they are small, as part of the description of the component.

System parameters This component type inherits the parameters describing the Shell Command component type. Additionally, the `command` parameter is replaced by the following:

Input		
	`lang`	the scripting language interpreter to use.
	`embedded`	flag indicating whether the script is embedded or it is to be found in an external file.
	`script`	the embedded script, it may also contain parameter placeholders that will be expanded before the script is passed to the interpreter.
	`scriptref`	the filename where the script is stored. In this case the script cannot contain any parameter placeholders as it can only receive input data through the command line.
	`args`	the command line arguments to pass to the script.

The mappings for Control, Scheduling, Data, and Failures are equivalent to the ones of the previously described Shell Command component type (Section 4.4).

4.6.2. SQL

A database query is also a useful component type for conveniently sending a set of SQL statements to an external database and, if applicable, retrieve the results of the query and store them in the output parameters.

With this component type, it becomes easy, for example, to write a process that stores persistently into a database table the data produced by services belonging to other component types. Conversely, a process can also be used to extract data from a database and process it with the services provided by other component types. This component type also provides the infrastructure to build a process to integrate data coming from different sources, some of which can be SQL database queries. As opposed to invoking external services or applications that interact with a database, using this component type may provide a faster development option, as the development of external database clients is not required. At runtime, to further reduce the execution overhead, JOpera's database adapter can pool shared database connections among all of its concurrently running processes.

System Parameters In order to describe an SQL script to be sent to a database we use the following system parameters:

Input System Parameters	`dbtype`	The database type. This information is needed to locate the appropriate JDBC driver.
	`server`	The address of the database server.
	`port`	The port of the database server.
	`user`, `pass`	The authentication information used to connect to a certain database.
	`database`	The name of the database to use.
	`script`	The script with the SQL statements to execute.
	`prepare`	An optimization flag indicating whether the statement should be prepared.
Output	`status`	The JDBC status after the query has been executed.
	`errormsg`	The JDBC error message, if any.
	`data`	The result of the query.

Control and Scheduling Given the client-server type of interaction, components of this type are executed synchronously. Considering a database replication scenario where a set of alternative database servers is available, the `server` address can be chosen dynamically by a scheduler.

Data There are several approaches to exchanging data with a database server through JDBC. With the usual placeholders mechanisms the values of the input parameters can be replaced directly into the SQL script before this is sent to the server. As an alternative, in case of prepared statements, the parameters of the SQL statement are passed using a positional encoding to the server: the order in which the input parameters are defined must correspond to the order in which the parameters are referenced in the SQL statement.

Also for modeling the results of a query, so that they can be stored into user output parameters, there are different approaches. Considering that it is possible to refer to each field of a tuple by name, and assuming that the database schema is known in advance, a simple approach is to name the user-defined output parameters of SQL components with identifiers corresponding to the fields of the resulting dataset. Nevertheless, how to efficiently encode the potentially large results of a query so that they can be used in practice in the rest of the process remains an interesting (and open) optimization problem.

Failures An SQL script may fail for various reasons, which are summarized by the value of the JDBC `status` parameter. To simplify error recovery, it is assumed that each SQL component is invoked within its own transaction.

4.7. XML Data Manipulation

In practice, Web services of a realistic complexity expect to receive large data structures as input messages and may also produce complex XML documents as result. Inside SOAP messages, such data structures are normally encoded in XML strings conforming to an XML schema instance [247], which is referenced by the Web service interface description. Messages returned from one service can only rarely be forwarded directly to another [15]. Instead, some form of XML Data manipulation is usually needed for transforming and adapting such message to a different data model, which may have different syntax or semantics [212]. Similarly, complex results of a Web service may need to be partitioned so that they can be passed on to many of the other services composing a process. Finally, XML data coming from several sources may have to be consolidated into a single result document to be returned to the user.

With JOpera there are two ways of approaching this problem:

1. It is always possible to leverage existing XML manipulation technologies, e.g., style sheet transformations (XSLT [243]) or the XML Path query language (X-Path [244]). This way, users familiar with these languages can embed XSLT transformation or X-Path expressions directly into a process by creating data filtering tasks which can be applied to the XML data in transit.

2. However, for a certain class of transformations, the JOpera Visual Composition Language can be used directly to model XML transformations in a visual way. These operations concern the encoding and decoding of XML complex-types. Furthermore, the split and merge operators for list-based iteration have been extended to support lists encoded in XML.

   Complex types A complex type is a record-like data structure which is composed out of elements of a certain data type, which can be simple (e.g., integer, boolean or string) or, again, complex [247]. For each complex type defined in the data model of a certain service's interface, we define two symmetric operations: *pack* and *unpack*. These operations are used to encode the XML representation of a data packet of a certain complex type (pack) and, conversely, to extract from its XML serialization each individual elements (unpack).

   These pack and unpack operations are automatically created by JOpera when importing the XML schema referenced by the Web service interface definition. A pack operation, for a certain complex type, has multiple input parameters, representing the elements of the complex type and one output parameter, which contains the encoded complex type. An unpack operation receives one input parameter, with the serialized complex type, and returns the values of its element in separate output parameters. The parameter types of the operations are copied from the original schema, allowing to statically check whether the pack and unpack operations are connected correctly. Furthermore, JOpera uses this type information to suggest the appropriate operation when

the user selects a parameter of a Web service having a complex data type. In case of data structures with nested complex types, we propose a modular, composable construction, where each complex type is encoded individually. The various packing operations can be then plugged together to form the final XML serialization.

4.7.1. XML Components

This category groups components used to manipulate data conforming to the XML format. Especially in the context of Web service composition, where XML data is the accepted standard format for data representation, it becomes important to easily access the XML transformation capabilities offered by these components types. In this area, a great variety of standards and new languages have been recently proposed. As representatives, we have chosen to model XPath queries and XSLT transformations, in order to show that it is possible to integrate such type of technologies within JOpera's component model. If necessary, it should not be too difficult to extend this category with other examples.

Although some of the system parameters depend on the specific type of XML component, there are some commonalities, concerning also the mapping of Control, Data and Failures, which are described in the following paragraphs.

System Parameters In general, an XML Component uses the following system parameters for processing an XML document:

`xmlin`	This system input parameter contains the XML data to be processed.
`xmlout`	The result XML data, if any.
`status`	The code which identifies whether an error occurred.
`errormsg`	The user readable information about the error.

Control and Scheduling XML Components are invoked synchronously, and no form of scheduling is supported.

Data The XML document to be processed is copied from the user-defined input parameters into the `xmlin` system parameter. The result is stored into the `xmlout` parameter, and can be then mapped the user-defined output parameters depending on the specific component type.

Failures XML Component can also fail. In general this may happen because the XML input data is syntactically incorrect or because there was an error while performing the actual operation. The `status` and `errormsg` system output parameters can be used to detect and debug the problem.

4.7.2. XPath Queries

An XPath expression is used to filter out of an XML document the required information [244].

System Parameters In addition to the parameters common to all XML components, the X-Path component type also uses the following one:

`script`	The X-Path expression to be applied to the XML data.

4.7.3. Style Sheet Transformations

Extensible stylesheet language transformations (XSLT) define a set of rules that, for example, are applied to an XML document to produce another XML document [243].

System Parameters In addition to the parameters common to all XML components, the XSLT component type also uses the following system input parameters:

`embedded`	flag indicating whether the style sheet is embedded or it is to be found in an external file.
`script`	the embedded style sheet.
`scriptref`	the filename where the style sheet is stored.

Data This component type also uses the same `xmlin` and `xmlout` parameters for transferring the input and output XML document. Furthermore, it is also possible to use parametric style sheets, where the output of the transformation still depends on the input XML document but it is controlled by the content of some additional parameters. Given that the parameters from within the style sheet are accessed by name it is possible to establish a one to one correspondence between the style sheet parameters and the user-defined input parameters of the component.

Example 4.2: Google Search

As an example of XML Processing with the JOpera Visual Composition Language and the XML components, we present how to retrieve a list of URLs from the results of a Google search. Since this WWW search engine's API has been published as a Web service [88], it is possible to import its interface definition, including its data model, into JOpera. In order to extract the required list of URLs, in the first example (Figure 4.7) we present a compact solution using X-Path queries. In the second example (Figure 4.8) we only use the JVCL's XML data manipulation features.

First of all, Google's search results are returned as a single data structure into the `return` output parameter. In Figure 4.7 we apply (in parallel) two data

filtering tasks (`FilterURLs` and `FilterCount`) to Google's `return` parameter. These tasks take an `xpath` expression and apply it to the content of their `xmlin` input parameter. The filtered data is returned in the `xmlout` parameters, which are then copied into the process output parameters.

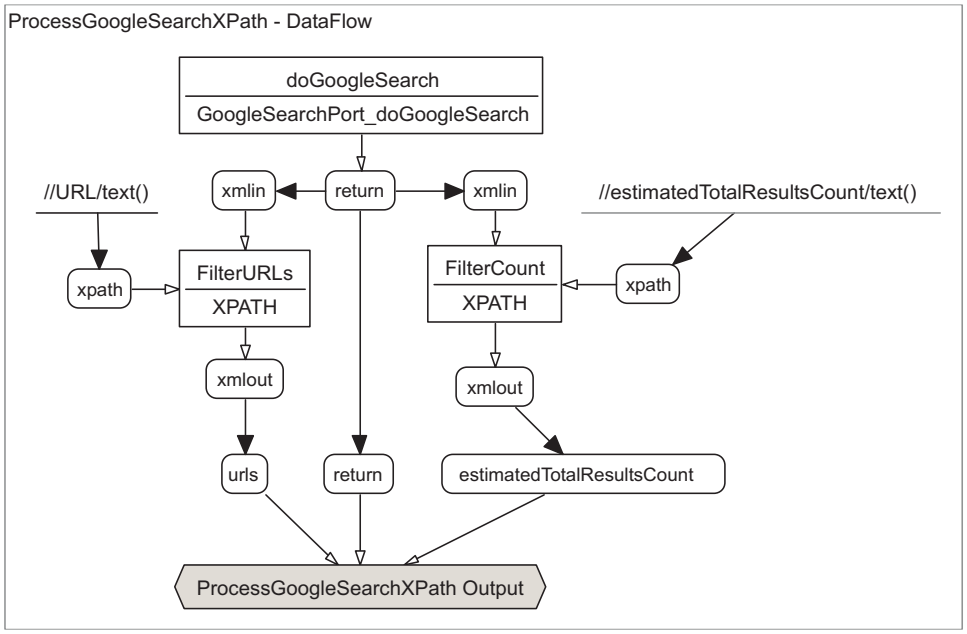


Figure 4.7.: *Example of XML Processing with the JVCL and X-Path.*

As an alternative (Figure 4.8), the less compact JVCL notation can be used to achieve the same result through different means. As opposed to modeling the data transformation with a declarative approach, we define operationally the data flow of the transformation.

More concretely, through the `Unpack_return` operation, it is possible to extract the component elements of the complex type returned by Google. These are, among others, the `searchTime` (indicating how long the query took), the `estimatedTotalResultCount` (indicating the estimated number of page hits) which is copied to the process ouput parameter with the same name, the `estimateIsExact` boolean parameter (indicating whether such number is exact) and, most important, the `resultElements` list. For simplicity, we have hidden the rest of the data elements of this type. In order to extract the list of URLs we iterate over this list with the split operator. For every element, contained in the `resultElements_item` parameter, the `Unpack_resultElements` operation returns the value of its content, including the desired `URL`. By applying the merge operator, the various values of the `URL` parameters can be now collected into the `urls` process output parameter.

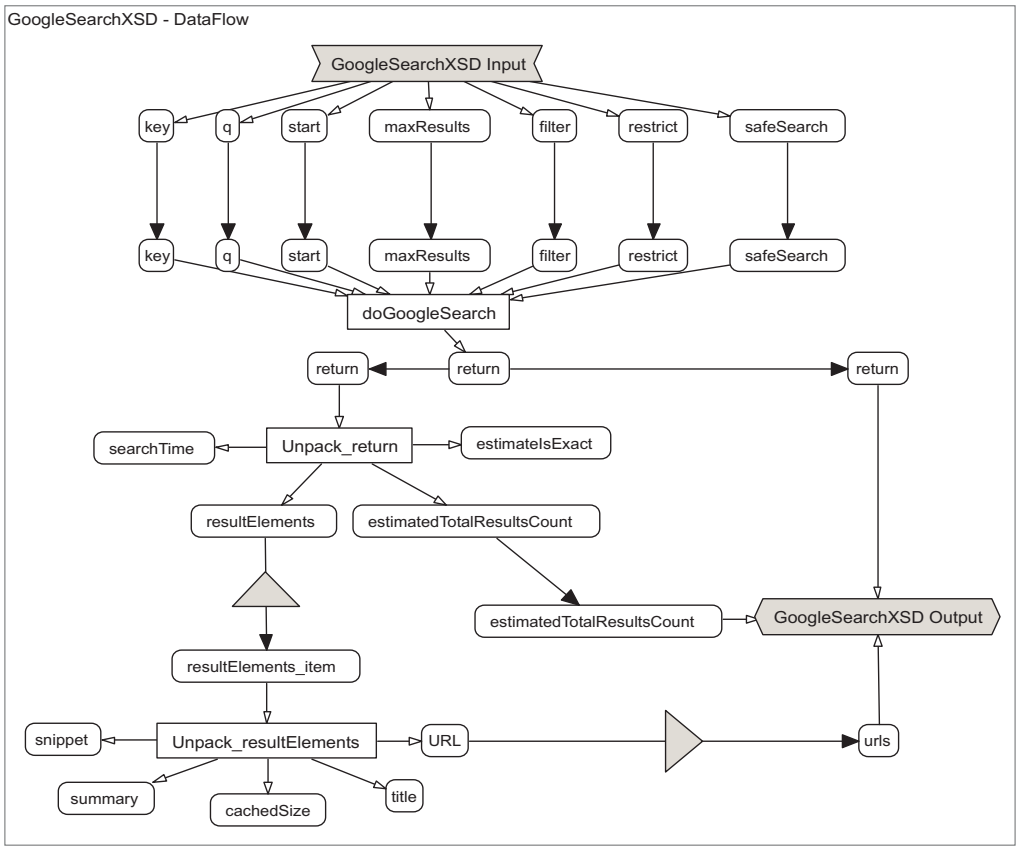


Figure 4.8.: *Example of XML Processing with the JVCL only*
This process retrieves a list of the URLs returned by a Google search.

By comparing the two examples it should be noted that with JOpera it is possible to perform basic XML manipulations with the approach most appropriate for the user's level of experience. By choosing the X-Path based solution (Figure 4.7), the advanced user can perform the data filtering with a a compact, declarative notation. On the other hand, with the operational, pure JVCL approach (Figure 4.8), it is possible to leverage the compiler's type checker to guide the user in finding the correct visual solution.

Example 4.3: Mismatching Services Adaptation

In the previous example, we have shown how to extract some information out of an XML document representing a complex data structure. Here, we would like to continue with this type of scenario and show another example on how to use the JOpera Visual Composition Language and the XML manipulation components to program the mapping required to make two services with mismatching interfaces fit with each other.

This example illustrates how to convert postal addresses between a service which returns them using a Swiss format (defined by the XML Schema [247] snippet of Figure 4.9) to the US format (Figure 4.10) understood by another service.

Both services have been built to manipulate postal addresses, both services use an XML Schema to define their data model. The information is even encoded in XML and transferred between the two services using the same SOAP protocol. Unfortunately the two services cannot be interconnected directly, because their data models are different and unless a mapping between the two interface type definitions is designed and applied to the data in transit, the second service will reject the addresses received from the first one due to both syntactical and semantical incompatibilities [212].

```
<xsd:complexType name="Adresse">
 <xsd:sequence>
  <xsd:element name="name" type="xsd:string" />
  <xsd:element name="nachname" type="xsd:string" />
  <xsd:element name="strasse" type="xsd:string" />
  <xsd:element name="plz" type="xsd:int" />
  <xsd:element name="ort" type="xsd:string" />
  <xsd:element name="kanton" type="xsd:string" />
 </xsd:sequence>
</xsd:complexType>
```

Figure 4.9.: *XML Schema definition for the* `Adresse` *type.*

```
<xsd:complexType name="Address">
 <xsd:sequence>
  <xsd:element name="name" type="xsd:string" />
  <xsd:element name="street" type="xsd:string" />
  <xsd:element name="number" type="xsd:int" />
  <xsd:element name="town" type="xsd:string" />
  <xsd:element name="state" type="xsd:string" />
  <xsd:element name="zip" type="xsd:string" />
  <xsd:element name="country" type="xsd:string" />
 </xsd:sequence>
</xsd:complexType>
```

Figure 4.10.: *XML Schema definition for the* `Address` *type.*

As we can observe from Figures 4.9 and 4.10, the `Adresse` complex type does not match the `Address` type. Although some of its fields (elements) store equivalent information, e.g., the postal code, the fields are named differently (`plz` vs. `zip`). We also have a data aggregation conflict in the way the person's name is stored: using two fields (`name`, `nachname`) to differentiate between first name and last name, and only one field (`name`). In this case, the same field name (`name`) is used to tag incompatible information, if the fields with the same name would be considered to be matching, some information (the last name) would be lost. Furthermore, the first service returns Swiss addresses only, therefore there is no equivalent field to represent the `country` information,

required by the second service. Finally, the information about the street is also represented differently, using one field (`strasse`) in the `Adresse` type and two fields (`street`, `number`).

Although, at first sight, it may seem quite difficult to solve all of these problems, the data flow diagram with JOpera's solution is not too complicated (Figure 4.11) and all but the last incompatibility, concerning the `street` field, can be solved in an intuitive manner.

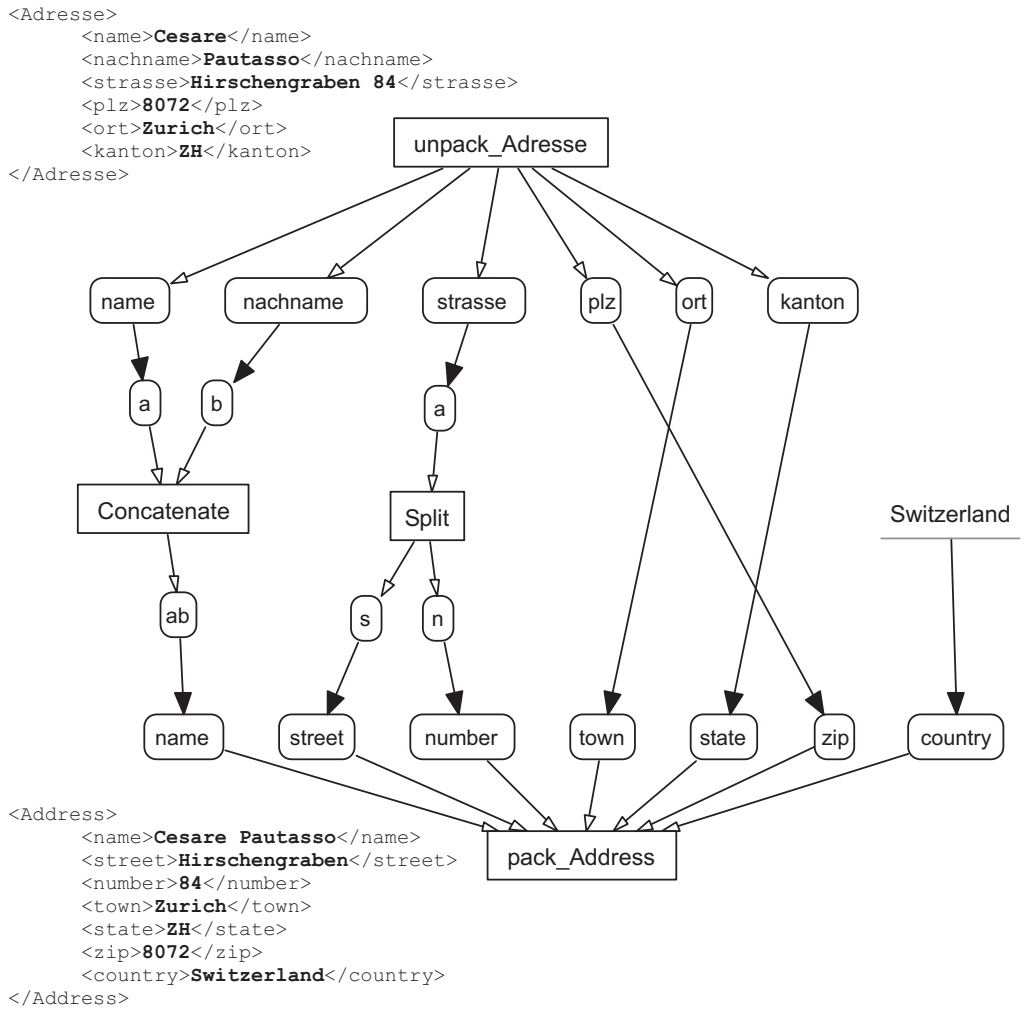


Figure 4.11.: *Visual mapping between two XML complex types*
This visual mapping converts postal addresses from a Swiss (top) to an American syntax (bottom).

The data flow graph of the mapping between the two complex types can be followed from top to bottom. For some pairs of fields (e.g., `plz` to `zip`), where we have a mismatch only at the description level, we can directly link the equivalent fields with a data flow connection. The `country` field, for which an

equivalent field is missing, can be bound to a constant value. The other fields have incompatibile values, therefore it is not enough to redirect the values to the appropriate field. Instead, the values need to be manipulated using string handling operators, which can be used to concatenate the values of two fields (`name`, `nachname`) into one (`name`), as well as to split the value of one field (`strasse`) into two (`street` and `number`). Now, in general, determining which part of a string value corresponds to a street name and which part corresponds to a street number may be rather difficult. For this example, and for the corresponding XSL transformation (Figure 4.12), we will assume a street number to be always stored at the end of the string, following the Swiss convention, and that it is separated from the street name by a blank character.

The example of JOpera's visual mapping of Figure 4.11 can be compared to the equivalent XSL transformation of Figure 4.12. Depending on the familiarity of the developer with this technology, using a visual mapping may be a more or less productive approach, when compared with the XSL-based solution. In principle, it is possible to take JOpera's visual representation and use it to generate the corresponding XSL code. For performance reasons we chose an alternative approach, which should maximize both the users' productivity (drawing a mapping can be faster than debugging XSL code) and the runtime execution's performance (JOpera's compiler generates Java executable code from the visual notation), as we will present in Section 8.2 on page 183.

```
<xsl:template match="/Adresse">
 <Address>
  <name>
     <xsl:value-of select="name"/>
     <xsl:text> </xsl:text>
     <xsl:value-of select="nachname"/>
  </name>
  <street>
     <xsl:value-of select="substring-before(strasse,' ')"/>
  </street>
  <number>
     <xsl:value-of select="substring-after(strasse,' ')"/>
  </number>
  <town>
     <xsl:value-of select="ort"/>
  </town>
  <state>
     <xsl:value-of select="kanton"/>
  </state>
  <zip>
     <xsl:value-of select="plz"/>
  </zip>
  <country>
     <xsl:text>Switzerland</xsl:text>
  </country>
 </Address>
</xsl:template>
```

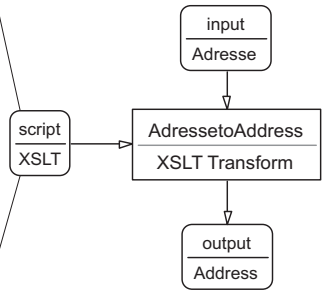


Figure 4.12.: *Equivalent XSL mapping*
An alternative, equivalent representation based on XSL of the visual mapping in Figure 4.11 (left). The style sheet can also be invoked from a JOpera process (right).

Performance, however, should not be the only criteria for comparing the two solutions. As indicated in Figure 4.11, it is possible to recognize in the visual mapping the following three stages: 1) The `unpack_Adresse` operator *extracts* information from the original syntax of the source service. 2) A visual *mapping* based on data flow arrows and operators is used to define the transformation appied to such information. 3) The `pack_Address` operator *formats* the results according to the syntax of the destination service's representation. As discussed in [178], it is important to keep these aspects separated. This way, the semantic-level mapping can be specified in terms which are independent of the actual syntax-level formatting of the information. Since the XSL transformation of Figure 4.12 does not support such clear distinction, it remains more difficult to understand and maintain.

4.8. System Components

These components represent basic facilities and services provided within the JOpera system that can be invoked from a process without any external dependencies. They include a testing mechanism for echoing back the same input data as output, the internal mechanism used for calling sub-processes and a set of components exposing part of the JOpera API (Section 7.5), so that it becomes accessible from within a process.

4.8.1. Echo

The Echo system component type has been introduced for testing purposes, but can also have useful applications. For example, the previously mentioned (Section 4.7 on page 65) pack and unpack operators used to encode and decode XML complex data types are implemented using this system component.

Essentially, the components of this type repeat the received input data back as output, and do not involve the interaction with any external service provider.

System Parameters This component type has one system parameter with two manifestations:

`input`	This input system parameter can be assigned with the value to be returned.
`output`	By definition, this output system parameter contains the same value as the `input` parameter.

Control and Scheduling This type of components is executed synchronously.

Data Considering that parameter placeholders can be used to build the `input` system parameter and that the content of the `output` parameter is interpreted according to the usual XML syntax, in practice it is possible to use this type of component to model n to m simple data transfers where a set of n parameters is mapped onto a different set of m parameters, where $m \leq n$.

Failures Components of this type only fail if the `output` data parameter cannot be parsed in order to extract the specified user output parameters.

4.8.2. Process Invocation

This type of system components is used behind the scenes to implement the sub-process construct of the JOpera Visual Composition Language. Although syntactically different, a sub-process is semantically equivalent to an activity referring to this kind of component, which represents the invocation of a process.

System Parameters

Input System Parameters	`procname`	This parameter contains the name of the process to be invoked. In most cases its value is fixed at compile time. However, to model a form of late binding, it is possible to dynamically replace its value at runtime.
	`spawn`	This flag indicates whether the process should be started asynchronously. In case of asynchronous process calls, the invocation of the component completes as soon as the process has started. By default, process invocations are synchronous, i.e., they complete after the called process has terminated.
	`lookup`	This flag is used to ask JOpera to look for processes that have already been run with the same input parameters so that their results can be recycled and no new process needs to be started.
	`priority`	This parameter controls the scheduling priority of the process
	`numinstances`	This controls the number of instances of the process to be started. By default only one instance of a process is started wth a sub-process call. This parameter is used to perform scalability testing.
	`resources`	This parameter constraints the scheduling of the tasks within the invoked process to the specified set of resources.
	`delete`	If this flag is set, the process instance will be deleted automatically after it has completed its execution.
Output	`exception`	This system output parameter identifies the tasks that caused the failure of the process.
	`procid`	This parameter contains the `ID` of the invoked process. In case of asynchronous calls, it allows the caller to identify the process which is running in the background in order to interact with it.

Control and Scheduling Sub-processes are invoked asynchronously by queuing a "start process" request which contains additional information identifying the caller process. When the invoked process completes this information is used to notify the caller that the sub-process has completed.

Data Both input and output data between the sub-process and the invoked process are exchanged implicitly. By definition, a sub-process has the same user-defined input and output parameters as the invoked process. Therefore, there is no need to explicitly model this data transfer that happens automatically.

Failures The failure of the synchronously invoked process will always trigger the failure of the calling sub-process. In this case, the `exception` system output parameter can be used to identify the tasks that first caused the problem. An asynchronous process call will fail only if the process cannot be started, e.g., because its name is invalid.

4.9. Cluster Computing

In our work in the context of the BioOpera project [23], we used the JVCL language to model cluster (and grid) computations as processes, where the execution of each task involved the scheduling of the corresponding service invocation, in order to determine the optimal node of the cluster (or grid) to provide the computational service. As a natural generalization of this approach, in this section we present another type of components, related to cluster computing.

Cluster computing components model the ability to submit a computational job to a cluster of computers, or more precisely to the resource management and batch scheduling system which controls such cluster. Examples of these systems include Condor [150], the Sun Grid Engine [222] or the Portable Batch System [24] and BioOpera [23]. Mapping a service invocation to the submission of a job to one of these systems opens up the interesting opportunity of building grid-oriented processes that coordinate computations across heterogeneous cluster management systems, spanning across multiple sites and organizations [22].

Although some details may vary depending on the actual system, in general, such job submission involves the packaging of the computational job into a script and the description of the requirements of the job in terms of computing and storage resources in metadata associated with the job submission. As we have seen so far, such information can be modeled with system input parameters in a straightforward manner.

4.9.1. PBS

The Portable Batch System [24] is an example of a cluster resource management and scheduling system used to optimize between the resource utilization of the cluster and the turnaround time of the submitted jobs. Its interface is based on queues.

Each queue accepts jobs in form of scripts that contain both metadata (attributes) and the actual sequence of shell commands to be executed on one or more hosts of the cluster. The output produced by PBS jobs is typically stored into temporary files.

System Parameters This is the minimal number of system parameters that describe a PBS job submission[3].

`queue`	The name of queue of the batch scheduling system to which the job should be submitted.
`script`	Script which contains the attributes controlling the job and the commands to be executed as part of the job.
`stdout`	This system output parameter contains the output produced by the job.
`stderr`	The error messages printed by the job.
`pbsid`	The PBS job id.

Control and Scheduling Jobs are submitted asynchronously to a PBS queue, which is periodically polled to determine the state of the job. As an alternative, an email notification can be sent when the job has completed.

Data Typically PBS jobs exchange data through external files, therefore the user-defined input and output parameters contain the names of such files.

Failures As PBS only returns the completion notification for a job, it is not clear how to determine its outcome in a general way. However, if a problem occurs during the job submission, e.g., the selected queue is not available, this condition can be detected immediately.

4.9.2. BioOpera

System Parameters The BioOpera component type extends the UNIX component type with scheduling capabilities modeled by the following system parameters.

`resource`	The name of the resource on which the command should be executed.
`host`	The name of the host on which the actual execution has been scheduled.

[3] A more extensive model would explicitly include as system input parameters describing each of the job attributes defined by PBS (e.g., priorities, file staging, resource requirements, etc.). These attributes would be automatically encoded together with the command script into a job submission.

Example 4.4: Parallel Image Rendering

In this example we show how to use the JOpera Visual Composition Language to program a parallel computation used to speed up the rendering of ray-traced images with a cluster of computers. This computation has been built by composing a set of Shell Commands that are run through the BioOpera scheduler.

The structure of the `RenderImage` process can be reused for other, similar data parallel computations, where a large computational task is partitioned into smaller, independent units (or chunks) that can be then executed in parallel by submitting them to the cluster management system. Once all units have completed their results are merged for further post-processing.

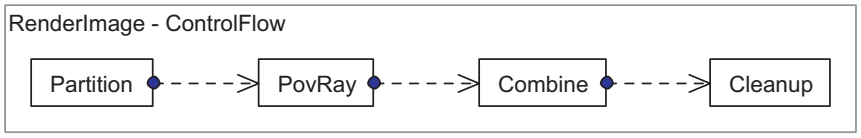


Figure 4.13.: *Control flow view of the* `RenderImage` *process*

As represented in Figure 4.13 the process is composed of four tasks, which are executed in sequential order: `Partition`, `PovRay`, `Combine`, `Cleanup`. For enhanced readability, the data flow graph is shown in two separate views Figures 4.14 and 4.15, the first covering the parameters of the first two tasks, and showing the parallel computation. The second shows the data flow of the last part of the process, where the results of the parallel computation are merged.

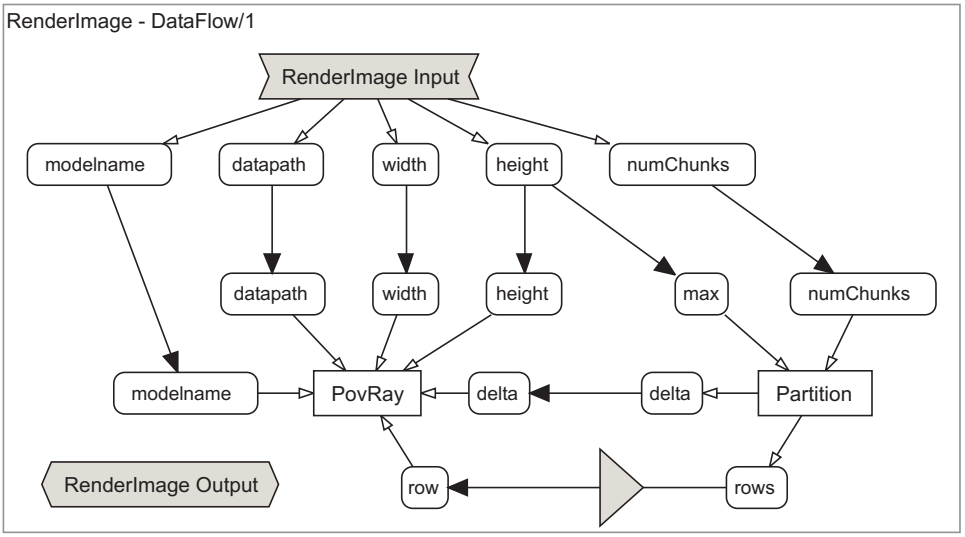


Figure 4.14.: *Data flow view of the first part of the* `RenderImage` *process*

The process receives the following input parameters: `modelname` and `datapath` contain the name and the location of the scene to be rendered; `width` and `height` contain the desired size of the resulting image. The `numChunks` pa-

rameter is used to control the number of partitions. As shown in Figure 3.13, the value of this parameter could be set automatically by JOpera's resource manager based on the current state of the cluster.

In this example the final image is partitioned into horizontal slices, each of which can be rendered in parallel. Other partitioning strategies are also possible, e.g., vertically or along a grid, but, considering the way image data is stored (along horizontal scanlines), they would make the merging step more difficult. The `Partition` task, invoked at the beginning of the process, uses the `height` of the image to produce the `rows` parameter, which contains the list of row indexes where each parallel rendering should start. The `delta` parameter contains the number of rows that should be rendered for each partition.

After the image `Partition` task has finished preparing the work to be done in the parallel part of the computation, for each `row` in the `rows` parameter, a parallel execution of the `PovRay` application [189] is started, as indicated by the split operator between the two parameters. This task receives as input the `datapath` and `modelname` parameters, indicating where it should read the scene description to be rendered. Furthermore, the size (`width` and `height`) of the final image, as well as the size (`delta`) of the horizontal slice are also passed to the task. The partially rendered images are written into the same `datapath` directory so that they can later be merged by the `Combine` task, which also receives the `datapath` and `modelname` input parameters (Figure 4.15) and uses them to look for the various slices of the image.

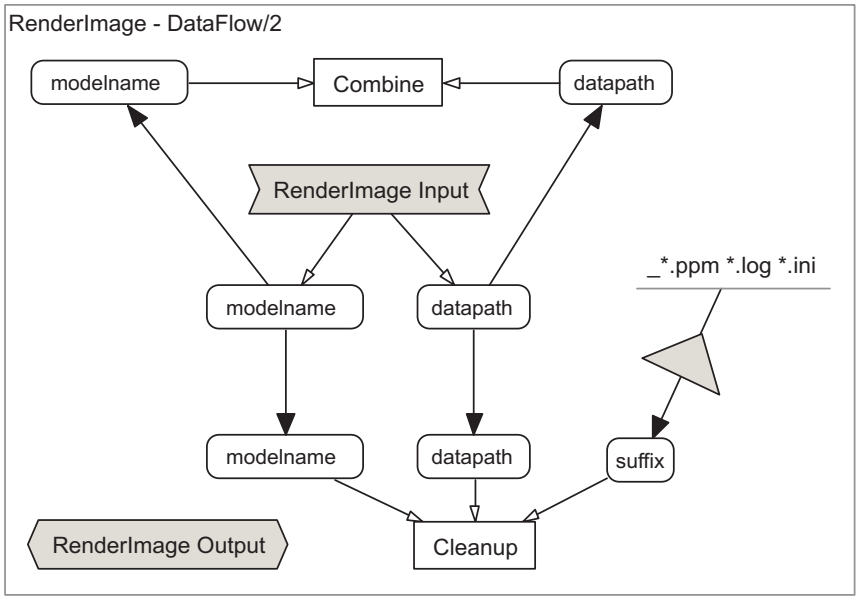


Figure 4.15.: *Data flow view of the second part of the* `RenderImage` *process*

Once the final image has been produced, a parallel instance of the `Cleanup` task is started for every different `type` (`_*.ppm`, `*.log`, `*.ini`) of temporary

file to be removed from the `datapath` directory. Given the typical number of partitions used and the number of temporary files created during the parallel rendering, it pays off to also perform the cleanup in parallel for each type of file.

The `RenderImage` process is an example of a process with a partially implicit data flow. First of all, the process does not model the transfer of image data between the rendering (`PovRay`) and merging (`Combine`) tasks. Instead, it is assumed that such data is exchanged through a file system shared among all nodes of the cluster. Therefore, these tasks need to follow the same file naming conventions, so that it is enough to give a directory path name in the common `datapath` parameter to be able to reconstruct which files need to be merged into the final result. As a consequence, the control flow diagram in Figure 4.13 cannot be automatically derived from the data flow of the process and the additional constraints between the `PovRay`, `Combine` and `Cleanup` tasks have been manually added.

4.10. Messaging Components

Message based interaction is the basis for building loosely coupled distributed systems out of services, which are invoked asynchronously. One of the interesting results of applying JOpera component meta-model to this type of components is that no extension to the composition language is required to distinguish between the synchronous and asynchronous invocations of services. Furthermore, thanks to this type of components it becomes possible to model the asynchronous cancellation of the execution of a process. Likewise, processes, which are normally used to describe the asynchronous interaction with services, can use these components to also implement such conversations.

In this context, we present two different types of components used for synchronously sending and receiving messages. The first one uses the SMTP [193] and the IMAP/POP [55, 169] protocols, collectively known as electronic mail. The second type of messaging component is based on the Java Message Service (JMS) specification [220].

In general, the model of these components has a similar structure, as it is necessary to provide addressing information (identifying the source and destination of a message, e.g., a *queue*), the content of the message and additional header information including, e.g., the priority of a message or timestamps and further identification information. In practice, we have chosen to classify these components in different types in order to present the developer with a model closer to the one used in the underlying messaging system.

4.10.1. Email

This type of component is used for sending and receiving electronic mail messages.

System Parameters The first set of parameters are input parameters used to identify the email server to use. Then, depending on the value of the `command` parameter, the next set of parameters are used as input (to send a message) or as output (to receive a message).

Input	`server`	The address of the email server.
	`protocol`	The protocol (SMTP, POP, IMAP).
	`user`, `pass`	If required by the protocol, the user authentication information to access the email account
	`id`	Optional message identification information.
	`command`	Whether to `send` or `receive` a message.
Input or Output	`from`	This system input parameter contains the email address of the sender of the message.
	`to`, `cc`, `bcc`	These parameters contains the email addresses of the recipients.
	`headers`	This parameter contains additional headers for the message, which specify, for example, the priority of the message or whether a return receipt is expected.
	`subject`	The subject of the message.
	`body`	The main text of the message.
	`attachments`	A list of attachments.

Control and Scheduling A message is sent or received asynchronously. When receiving a message, the invocation of this component returns after either a message has been received or the given `timeout` has expired.

Data The user defined input (or output) parameters can be mapped to both the body and the attachments of the message being sent (or received).

Failures Failures that prevent the message from being sent are detected from the corresponding SMTP status codes. Similarly, the `status` output parameter indicates whether a message has been successfully received within the given `timeout`.

4.10.2. JMS

This component type models the messaging services of the Java Message Standard specification [220].

The mappings for Control, Scheduling, Data, and Failures are very similar to the ones of the previously described EMail component type (Section 4.10.1).

System Parameters Like the previous example, some system parameters are used as input or as output depending on the value of the `command` parameter.

Input	`server`	The JNDI name of the JMS connection factory used to access the messaging system.
	`command`	Whether to `send` or `receive` a message.
Input or Output	`destination`	This parameters contains the queue or topic name used to identify the recipient of the message.
	`replyto`	This system input parameter is used to identify the sender of the message.
	`messageid`	Optional message identification information.
	`deliverymode`	This optional parameter specifies the reliability guarantees for the message, i.e., whether the message is sent best-effort or it is stored persistently.
	`headers`	This parameter contains additional headers and properties for the message.
	`content`	The content of the message, string encoded.

Example 4.5: Asynchronous Process Call

In this simple example we show how to use the messaging components to model the asynchronous interaction between two processes. The example is equivalent to a synchronous (i.e., blocking) sub-process call, however the two processes exchange data by sending messages and the sub-process construct is not used. Furthermore, the example can be easily generalized to model arbitrary multi-party interactions using different message queues.

Figure 4.16 shows the data flow of the `ClientProcess`. This process sends the value of its input parameter `c` as `data` on an `input queue`. At the same time, as there is no control flow dependency between the `Send` and `Receive` tasks, it waits to receive a message on the `output queue`. Once the `data` arrives, it is copied to its output parameter `f`, before the process finishes.

It should be noted that after the input message has been sent, there is no particular constraint that limits what the process may do. In fact, other tasks can be added to be executed independently of whether an output message has been received. Furthermore, in this simple example no correlation betweeen the input and output messages is enforced. If the `output queue` already contains messages before the `ClientProcess` is started, one of such messages could be received even before one is sent on the `input queue`. By using additional parameters of the messaging components in combination with reflection, is possible to associate the `ID` of the process to the messages in transit. Thus, only the messages related to a particular process instance will be processed by such instance.

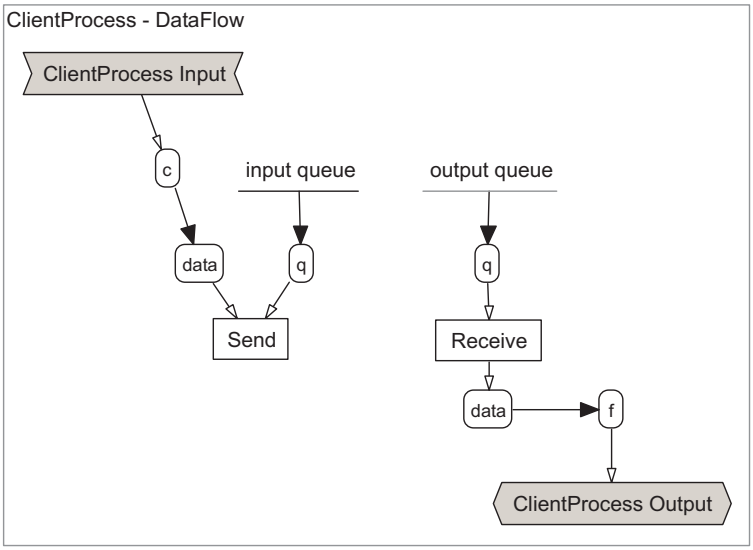


Figure 4.16.: *Data flow view of the* `ClientProcess`

Figure 4.17 shows what happens on the other end of the two message queues. Symmetrically, in the `ServerProcess`, messages are received from the `input queue`, processed and the results sent on the `output queue`. During execution, the `ReceiveRequest` task waits for an incoming message on the `input queue`. Once a message has been accepted, its `data` is passed to the `c` input parameter of the `Compute` task, which transforms it into the `f` parameter. Its value is copied into the `data` parameter of the `SendResult` task, which sends it in a message on the `output queue`. At this point the execution of one instance of the `ServerProcess` completes.

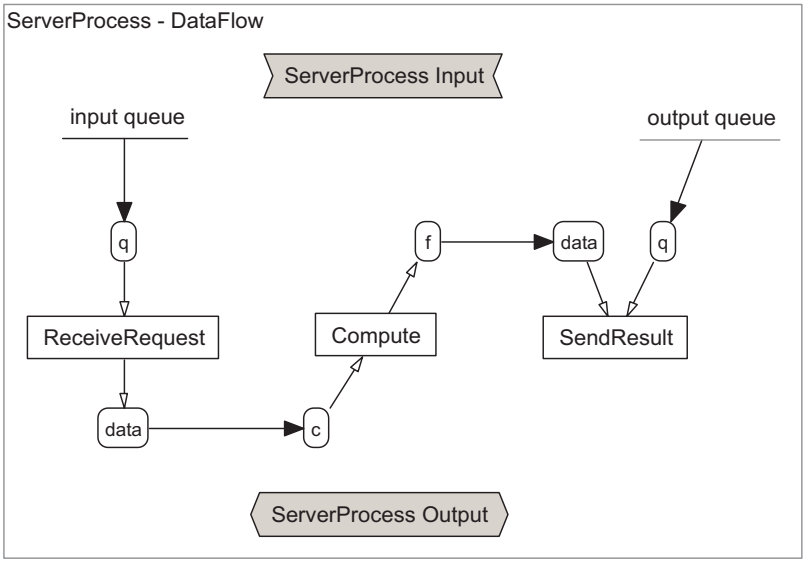


Figure 4.17.: *Data flow view of the* `ServerProcess`

4.11. BPEL Basic Activities

In this section we present how some of the basic activities defined as part of the BPEL4WS [112] specification can be modeled by a particular JOpera component type. This way, the functionality provided by these activities can be accessed from within a process without having to extend the JVCL language with new constructs. Out of the various BPEL basic activities we only need to model three ones (`wait`, `throw`, and `empty`) for the others (`assign`, `reply`, `receive`, and `invoke`) have an equivalent mapping to either a JVCL constructs (data flow bindings) or other component types (Messaging or Web services)[4].

System Parameters The three BPEL basic activities we are interested in modeling do not produce any data as output, therefore only one input system parameter will suffice:

`bpel`	BPEL representation of the basic activity to be executed.

Control, Data, Failures Depending on the content of the `bpel` parameter, different actions are carried out synchronously (without scheduling). Where applicable, input data can be inserted in the BPEL representation through the usual parameter placeholder mechanism.

```
<throw faultName faultVariable/>
```

The invocation of this component will always fail. The `faultVariable` attributes indicates from which user-defined input parameter the additional information about the fault should be taken.

```
<wait (for|until)/>
```

The invocation of this component will return after the given relative or absolute timeout has expired.

```
<empty/>
```

The invocation of this component will return immediately without any effect.

[4]This decision is explained in the presentation of the mapping of BPEL4WS to JOpera's process model in Section 6.4 on page 125.

4.12. Workflow Tasks

This type of component models human-oriented tasks that are usually assigned to a person for being taken care of. Traditional business process models included this component type as the only basic form of activity to be composed into a workflow. In JOpera, without including these component types, the interaction between a process and the user running it normally happens only when the process starts, as the user supplies the input parameters of a process. By using the debugging environment, a user can always interact with a running process to resolve unexpected situations. However, in some cases, it is useful to explicitly model in a process a point in the execution where the interaction with the user is always required. For example, a human operator may have to check the partial results of a process and use them to make a decision on how the rest of the execution should proceed. Likewise, exception handling tasks may correspond to the invocations of the services provided by human troubleshooters.

System Parameters As the user-defined input and output parameters are also visible to the person handling the workflow task, the system parameters are used for describing what needs to be done and controlling who should do it.

`task`	This input system parameter contains a textual description of what needs to be done as part as the workflow task.
`role`	This parameter identifies the set of people within a certain organizational unit that can and should handle the workflow task.
`errormsg`	This output system parameter contains the user-provided description of what caused the failure in the execution of the task.
`operator`	The operator to whom the task has been assigned.

Control and Scheduling Workflow tasks are executed asynchronously through a so-called Worklist handler which uses the `role` parameter to schedule the execution of the task among the available human resources.

Data When the task needs to be executed, within the active worklist item, in addition to its basic description, the user can also directly read the value of the input parameters and fill in the values for the output parameters.

Failures Similar to other component types, also workflow tasks can fail either because nobody has executed them within a certain time or because during their execution a problem occurred.

4.13. Discussion

With all of the examples of different component types presented in this chapter, we have attempted to show the flexibility of JOpera's component model, which provides the developer with a choice of several different mechanisms for describing and accessing the services to be visually composed into a process. More specifically, thanks to our simple component meta-model, from the perspective of the developer, the heterogeneity of the available component types does not sacrifice the *unity of interface* principle, where the same user interface should be used for the same task, regardless of how it is implemented behind the scenes [266]. In our case, regardless on what is the actual mechanism used to invoke a service, its representation within a process, both in terms of its visual syntax and of its parameter based interface, remains the same for all types of components. In Chapter 7, while presenting JOpera's architecture, we will show how the support for heterogeneous component types has been designed taking into account the trade-off between efficiency and flexibility, i.e., between keeping the service invocation overhead small and conveniently providing support for a wide range of component types.

On a different level, it should be also noted that, in general, we have been modeling services as components which are readily available, ignoring their dependencies. On the one hand, it is the job of the service provider to ensure the satisfaction of such *internal* dependencies, which may go from solving installation and deployment issues all the way to the management of the underlying hardware infrastructure [225]. Therefore, as opposed to traditional component based software engineering, the client should only be interested in modeling what the services provide and may disregard their requirements when composing them.

On the other hand, from JOpera's point of view, component definitions introduce an *external* dependency between the definition of the component itself and the external service providing the actual functionality. This can have repercussions when process definitions and the included component definitions are moved to a different process execution environment, i.e., processes become unusable because the services they require are missing in the new environment.

First of all, in JOpera, the clear separation between processes and their component service definitions helps to make the process descriptions themselves independent of such changes, as the processes are defined in terms of service interfaces only. The required adaptation work is thus limited to the component definitions only.

More in detail, for component types modeling globally accessible services with a remote implementation, e.g., Web services, this dependency is quite small as it amounts to the given URL of the WSDL document and to the interface definition of the service contained in such document. As long as both of these pieces of information do not change, it is still possible to modify the service's implementation or move the location of its provider, without invalidating the component definition.

For other component types, e.g., UNIX applications, the bond between the component definiton and the external application is stronger. If, for example, the process and the included component definitions are ported to a different execution environment, it must be ensured that the referred applications are locally accessible and

that all of their dependencies (e.g., in terms of file naming conventions) are satisfied. Depending on the application, this may amount to a simple reinstallation or it may entail more difficult configuration work.

In case of components such as Java scripts or other component types that support the embedding of the service's implementation within the component definition, such dependencies to external artifacts are minimized. As long as the required version of the scripting language interpreter can be found, the scripts can be immediately executed in the new environment.

To address these process portability issues, thanks to the information contained in JOpera's component model, when deploying a process into a new execution environment, it is possible (up to a certain extent) to actively check that all of its dependencies are satisfied and that all of the required services are available and accessible.

As a final note we would like to emphasize that, relaxing the constraints on the type of components that can be composed can contribute to the generality and simplicity of the composition language. Since all JOpera component types have the same parameter-based interface, no ad-hoc language construct is needed to discriminate between the invocation of services of different types (for example, it is not necessary to distinguish with different language constructs between synchronous invocation or asynchronous, message based interaction). If it becomes necessary to access information dependent on the specific component type, reflection through the use of system parameters can be applied[5]. Still, also this mechanism is based on a consistent visual syntax, uniform across all component types.

All in all, we believe that the possibility of choosing (wisely) between the use of Web Services or other kinds of services can be of great value, as the most appropriate component type in terms of performance, security, reliability and convenience of use can be chosen.

[5]See Section 3.7 on page 34 for more information and some examples on using system parameters with reflection in the JVCL language.

5. Opera Modeling Language

In this chapter we present the definition of the the Opera Modeling Language (OML), the process modeling language used by the JOpera system. The Opera Modeling Language is a natural evolution of the Opera Canonical Representation (OCR) first described in [98]. As opposed to the text-oriented syntax of OCR, the Opera Modeling Language uses a syntax based on XML and it includes several additional features.

First of all, OML is the foundation for the JOpera Visual Composition Language (JVCL), the visual process modeling language of the JOpera system described in Chapter 3 and first presented in [187]. All of the visual elements of the JVCL language and the corresponding non-visual elements (processes, tasks, parameters, and so on) are stored in XML documents with the format defined by OML.

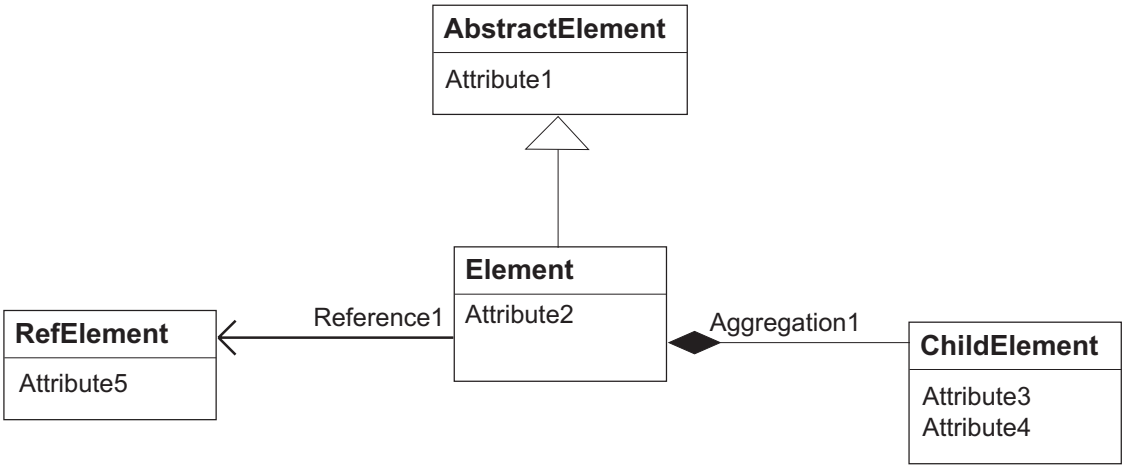
Second, an OML document stores separately the processes, which define the *composition*, from the programs, which defines the *components*. With this, a library of program definitions can be built and reused within multiple processes. Furthermore, an OML document also contain the model of the component types that can be invoked from JOpera, as presented in the previous chapter.

Finally, we will show in the next chapter that OML also defines an *executable* process model, as processes written in OML can be compiled to several executable representations.

5.1. Meta-Meta Model

In order to improve the understandability of the following description of the Opera Modeling Language (OML) in this section we would like to briefly introduce our notation and describe the modeling techniques used in the rest of the chapter. An OML document is an XML document, therefore it can be formally described using a Document Type Definition or an equivalent XML Schema [247]. Although such XML Schema is readily available in Appendix A, in this chapter we would like to present the Opera Modeling Language in a clear, more readable form. Furthermore, at the time OML was first designed, XML Schema had not yet come into existence and we had to follow our own (very simple) data modeling approach based on the following ideas.

First of all, the overall structure of an OML document and the relationships between its elements can be represented using a UML class diagram [176], where each class corresponds to an element and each attribute of a class corresponds to an




```
...
<Element Attribute1="" Attribute2="" Reference1="key1">
  <Aggregation1>
    <ChildElement Attribute3="" Attribute4=""/>
    <ChildElement Attribute3="" Attribute4=""/>
  </Aggregation1>
</Element>
<RefElement key="key1" Attribute5=""/>
<Element Attribute1="" Attribute2="" Reference1="key1"/>
...
```

Figure 5.1.: *Mapping UML to an XML document. Example of the relationship between the syntax of a UML class diagram (above) and the corresponding XML document structure (below) as it is defined in our Meta-Meta model. For clarity, the values of most attributes have been omitted.*

element's attribute (Figure 5.1). Moreover, in the UML class diagram we use three relationships: inheritance, aggregation and reference. The mapping between these class relationships to the structure of the XML document is defined as follows:

- We use inheritance for improving the clarity of the UML diagrams, as the common attributes and the reference and aggregation relationships shared among multiple classes can be abstracted into one. Thus, we can avoid describing the same relationships and attributes more than once. In the actual document only the concrete classes are instantiated into the corresponding elements, where all the attributes and relationships of the ancestor elements have been moved downwards along the hierarchy and inherited by the leaf elements.

- Aggregation, representing $1 \rightarrow N$ relationships between XML document elements, corresponds to the nesting of such elements inside one another. Furthermore, the nesting of elements happens through a single container node, which is a child of the element corresponding to the container class and con-

```
<xs:schema>
 <xs:element name="AbstractElement">
  <xs:complexType>
   <xs:attribute name="Attribute1"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="Element">
  <xs:complexType>
   <xs:extension base="AbstractElement">
    <xs:attribute name="Attribute2"/>
    <xs:attribute name="Reference1" type="xs:IDREF"/>
    <xs:all>
     <xs:element name="Aggregation1" minOccurs="0" maxOccurs="1">
      <xs:sequence>
       <xs:element ref="ChildElement" minOccurs="0" maxOccurs="unbounded">
      </xs:sequence>
     </xs:element>
    </xs:all>
   </xs:extension>
  </xs:complexType>
 </xs:element>
 <xs:element name="ChildElement">
  <xs:complexType>
   <xs:attribute name="Attribute3"/>
   <xs:attribute name="Attribute4"/>
  </xs:complexType>
 </xs:element>
 <xs:element name="RefElement">
  <xs:complexType>
   <xs:attribute name="key" type="xs:ID"/>
   <xs:attribute name="Attribute5"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 5.2.: *Mapping UML to an XML schema. XML Schema corresponding to the example UML class diagram of Figure 5.1.*

tains all aggregated elements as children. This way, it is possible to efficiently navigate the document structure and discriminate between elements of different types nested inside the same parent element. The name of the container element corresponds to the name of the aggregation relationship.

- References are simply used for modeling $M \rightarrow 1$ relationships, where the nesting of document elements would not be satisfactory. Thus, an element e can reference another element f, which can be found anywhere in the document, by storing the unique identifier of f in one of its attributes. The name of such attribute corresponds to the name of the UML reference relationship. The type of this attribute is denoted with "ref:f" to indicate that the value of the attribute is restricted to the keys identifing elements of the referenced type f.

As an example of how the previous rules can be applied, Figure 5.1 shows an XML snippet, whose structure satisfies the UML class diagram above it. The XML Schema corresponding to the UML class diagram is also shown in Figure 5.2.

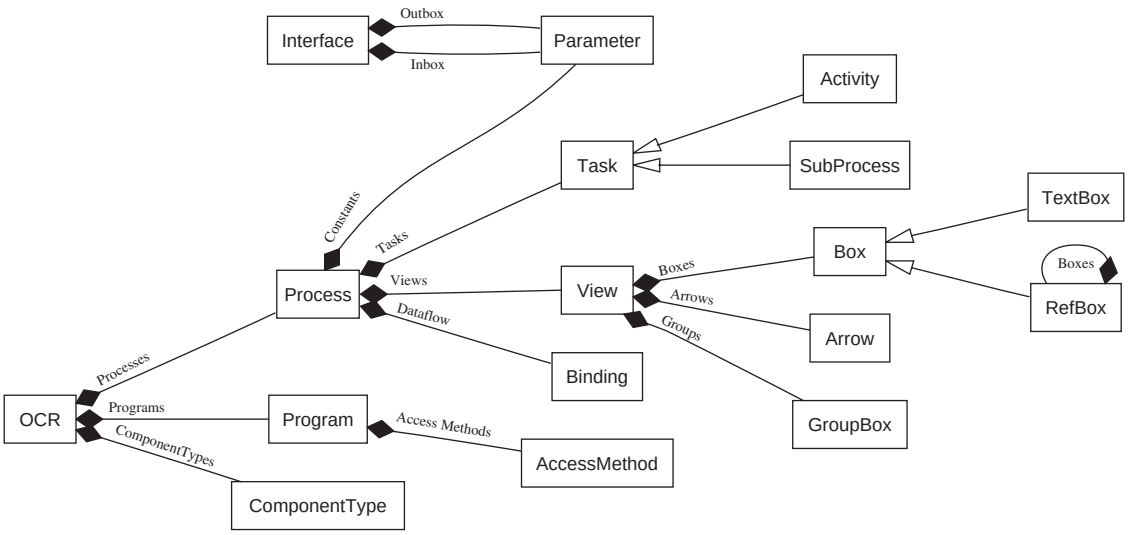


Figure 5.3.: *Summary of the aggregation relationships between OML elements. For clarity also selected inheritance relationships to concrete classes have been included. This UML class diagram gives a good overview over the structure of an OML document. Starting from the Root (OCR) element, it shows where the other elements are located with respect to each other.*

First of all, it should be observed that all of the nodes in the XML document have tags matching the names of the UML classes. Also in the corresponding XML Schema, the set of declared elements matches the UML classes. In the case of the `AbstractElement` class, there is no corresponding document node shown in the example. Although it would be possible to extend the meta-meta model with constraints that limit which UML classes can generate document nodes, for the purposes of describing the Opera Modeling Language we just imply that not necessarily all of the UML classes in the model will correspond to XML nodes, but all of the leaf classes in the inheritance tree will. More concretely, in the example the `Element` class inherits the first `Attribute1` from its ancestor `AbstractElement`. The `Aggregation1` relationship between the `Element` and the `ChildElement` classes corresponds to having one or more `ChildElement` nodes nested inside the same `Aggregation1` node, which is found inside the first `Element` node of the example. As specified in the XML Schema of Figure 5.2, for each aggregation relationship there can be only up to one of such container elements. The `Reference1` attribute of the `Element` nodes corresponds to the `Reference1` relationship in the class diagram. The value of this attribute (`key1`) is used to establish a unidirectional link between the two `Element` nodes and the `RefElement` one. As opposed to aggregation, where multiple children nodes can be nested into the same parent, by using this reference-based mechanism together with keys that uniquely identify nodes, it is possible to have many nodes referencing the same node from different locations in the document.

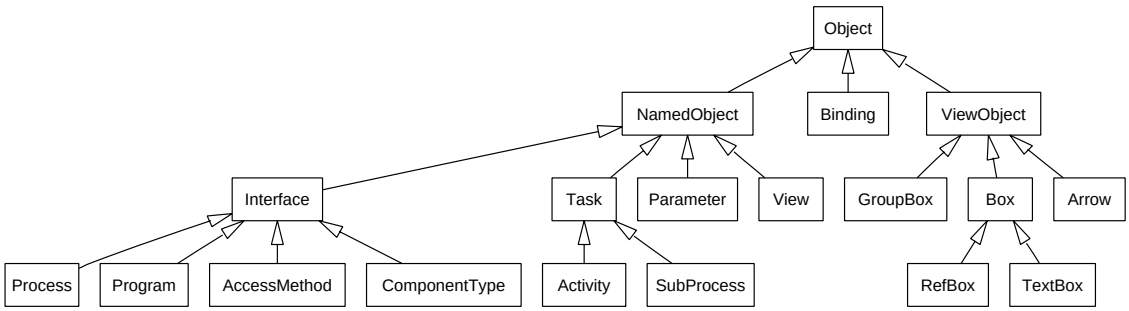


Figure 5.4.: *Summary of the inheritance relationships between OML elements. This UML class diagram illustrates how the common attributes and the shared reference and aggregation relationships of the document elements have been structured in a single-inheritance tree. The Object, NamedObject, ViewObject, Box, Task and Interface elements are abstract and do not appear in an OML document.*

5.2. Structure of the Opera Modeling Language

Following the meta modeling approach discussed in the previous section, the overall structure[1] of an OML document can be formally illustrated with the UML class diagrams in Figures 5.3 (Aggregation relationships), 5.4 (Inheritance tree), 5.5 (Reference graph) and 5.6 (All relationships together). It can also be described as follows.

The root element (`OCR`) of an OML document can contain a set of Process, Program and ComponentType definitions (Figures 5.3 and 5.7). In practice, the program definitions and the required component types declarations are usually defined once and included from separate OML documents.

The external Interface of a Process is defined by a set of input and output parameters. Internally, a process contains the list of its component tasks (Activities and SubProcesses), the data flow (Parameters and Bindings) and control flow graphs, as well as their visual representation (Views). The data flow graph is stored as a set of bindings between parameters or constant values. As the JVCL supports multiple, overlapping views over the same data flow graph, the data flow graph is also explicitly aggregated in the list of bindings of the process for efficiency reasons. On the contrary, only one view over the control flow graph of a process is allowed. Therefore, it is not necessary to store the control flow graph separately from the view representing it.

In an OML document, nested inside Processes, the data flow and control flow Views are represented as annotated graphs, i.e., they are composed of nodes (Boxes) and edges (Arrows) linking pairs of nodes. In addition to the graph's topology, the View elements include additional layout information to store the two dimensional position and size of the graph objects. Moreover, it is possible to group together some elements of a view (ViewObjects) in order to constrain the automatic layout algorithms that can be applied to the graph. The purpose of the most important

[1]Each OML element will be described in detail in the following section

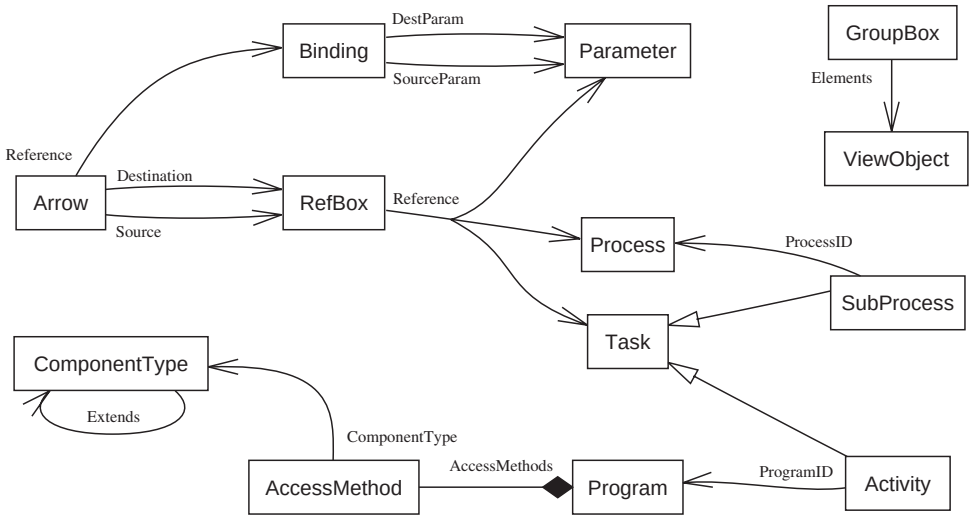


Figure 5.5.: *Summary of the reference relationships between OML elements. This UML class diagram shows how the OML elements refer to one another. In particular, the visual elements (Arrow and RefBox) refer to the model elements they visually represent. Moreover, references are also used to model the edges of the control and data flow graphs. Each edge is modeled with a pair of references, linking, for example, Arrows to the pair of RefBoxes between which a connection should be drawn. Reference are also used to model the relationship between the Activity and the Program to be invoked and, similarly, between the SubProcess and the Process to be called.*

elements of a View (RefBox and Arrow) is to visually represent other elements of the process. More precisely, a box may represent, the container process, a task or one of their parameters. Similarly, in a data flow View, an arrow linking two boxes represents a data flow binding between the two parameters represented by the boxes, and so on. As multiple elements of a view can represent the same process element, we use a Reference relationship in the UML class diagram to model the relationship between an element of the process model and its corresponding visual representation. As previously explained, in the OML document this amounts to storing the identifier of the referenced process element into the `Reference` attribute of the visual element (Figure 5.5).

The Program elements store the set of available component services that can be invoked with an Activity of a Process. Similar to Processes, also the Interface of programs is composed of a set of input and output parameters. Furthermore, a program can contain multiple Access Methods which define different, alternative ways to invoke the functionality provided by the service. Access methods also have input and output parameters, conforming to the template defined in the referenced ComponentType. By definition, the input and output parameters of an access method and the ones belonging to the corresponding component type are considered as system parameters.

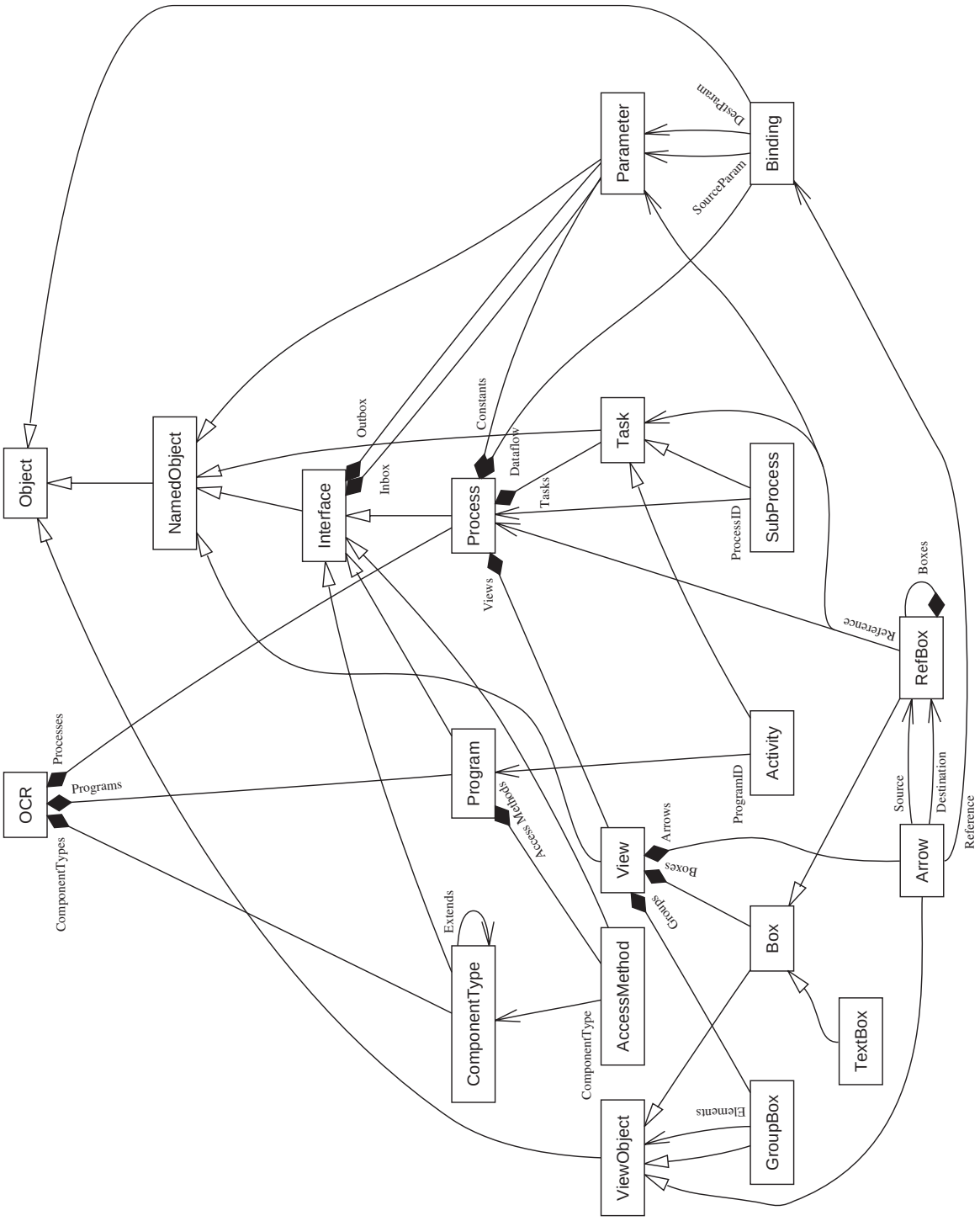


Figure 5.6.: *Complete UML class diagram of the OML elements. For completeness, we include also this UML class diagram with all relationships between all elements of an OML document.*

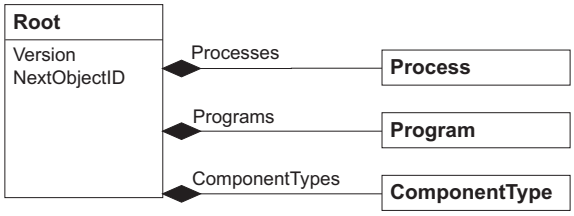


Figure 5.7.: *Basic structure of an OML document. An OML document is composed of Processes, Programs and Component Type definitions.*

5.3. Elements of the Opera Modeling Language

After giving an overview over the structure of an OML document based on UML class diagrams, in this section we continue with a more detailed description of each element of an OML document. This description includes, as a reference, the list of all attribute of each element, as well as all aggregation relationships for a certain element, i.e., the definition of what are the allowed children of a certain element and how they are grouped together. Sometimes, for attributes having a limited set of values, we also include the enumeration of the possible values. In addition to the basic description of the document elements and attributes, we have attempted to add a more precise explanation, which should motivate the design decisions that have been taken.

5.3.1. Root Element

The **Root** (`OCR`) element of an OML document (Figure 5.7).

Attributes of the Root element:

Name (`Tag`)	Type	Description
Version (`VER`)	String	The Version of the OML Format used in the file.
NextObjectID (`MAXID`)	Integer	As previously discussed, each document element should be uniquely identified, so that it can be referenced by other elements. To generate the necessary key values, we have chosen a simple strategy based on incremental numbering. For efficiency reasons, the root document element stores the global counter for determining the `OID` of the next element which is about to be added to the document.

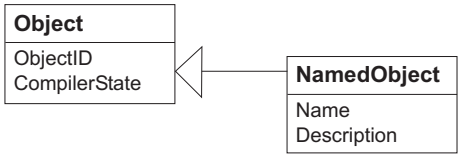


Figure 5.8.: *Abstract Elements: Object and Named Object. All OML Elements inherit their basic attributes from these two elements.*

Content of the Root element:

Name (`Tag`)	Element	Description
Processes (`PROCS`)	Process	Container for the processes defined in the document.
Programs (`PROGRAMS`)	Program	Container for the programs defined in the document.
ComponentTypes (`COMPS`)	ComponentType	Container for the component types defined in the document.

5.3.2. Abstract Elements

In order to avoid repetitions, the abstract elements are used to list the attributes and the aggregation and reference relationships which are common to the rest of the elements (Figure 5.8).

The **Object** element – Any object with an ID. If not specified, all other elements extend this basic one, as every element in an OML document must be uniquely identifiable.

Attributes of the Object element:

Name (`Tag`)	Type	Description
ObjectID (`OID`)	ID	The document-wide, unique identifier of the object. Typically a string formed by appending an integer counter to the element's tag. This is the key attribute, whose value can be referenced by other document elements.
CompilerState (`CS`)	CompilerState	The state of this object with respect to the compiler, i.e. whether the object is commented out and should be ignored or whether some error regarding the object has been found after checking the OML document for consistency.

Values of the CompilerState attribute of type CompilerState:

Value	Description
Ok	By default, the information of the element of the model is correct.
Comment	This element should be ignored by the compiler as it has been commented out by the user.
Error	An inconsistency that prevents successful compilation has been detected by the model checker, the specific type of inconsistency depends on the actual element. For example: at least two elements inside the same container have been given the same name or an activity does not reference any valid program.
Warning	The model checker has found some condition that can potentially lead to execution errors at runtime. For example, a parameter was not connected in the data flow graph.

The **NamedObject** element extends the Object element – It represents any element with a name and an optional description.

Attributes of the NamedObject element:

Name (`Tag`)	Type	Description
Name (`NAME`)	String	The name of the element, as it is shown to the user. In order to have a consistent document, the name should also uniquely identify the object within the enclosing container element. In case of duplicate names it is not always possible to compile the OML document into executable form, e.g., as duplicate names may lead to ambiguities in the control flow graph. However, considering that internally only the `OID` attribute only is used to store references between elements, having duplicate names does not affect the internal consistency of the OML document.
Description (`DESC`)	String	Optional description of the element. It can be used to store user provided comments associated to the element.

The **Interface** element extends the NamedObject element – This element represents the interface of any process element which can exchange data. The interface of a Process or a Program consists of input and output lists of user-defined parameters. In case of ComponentTypes and AccessMethods, such parameter lists are used for defining system parameters.

Attributes of the Interface element:

Name (`Tag`)	Type	Description
Author (`AUTHOR`)	String	The name of the user who has written the interface definition. As interfaces represent the smallest language element (a Process definition or a Program and Component declaration) that could be written by a certain developer, this common attribute is listed in this abstract element, as opposed to the NamedObject element.

Content of the Interface element:

Name (`Tag`)	Element	Description
Inbox (`INBOX`)	Parameter	Container for the input parameters.
Outbox (`OUTBOX`)	Parameter	Container for the output parameters.

5.3.3. Process Elements

This section describes the properties of a Process and its components: the Tasks, which can be either Activities or SubProcesses (Figure 5.9).

The **Process** (`PROC`) element extends the Interface element – A Process contains a number of tasks connected by data and control flow graphs, which are defined visually. In JOpera, a process is the smallest executable unit of composition. It describes how a set of service invocations are composed together. Through the SubProcess construct, processes can also be directly reused as components within other processes. Similarly, processes can be published as Web services for external reuse. To model and control its level of reuse, a process has the following attributes.

The **Task** element extends the NamedObject element – A task represent any component of a process: a basic step in the computation modeled by a process. The description of how such steps depend on each other is also part of the content of a task. More precisely, its Activator and Condition attributes model the basic control flow dependencies linking a task to its predecessors. The remaining attributes[2] are used for describing more advanced scheduling and synchronization options, in case the task belongs to a list-based loop, i.e., it is found within a pair of split and merge operators. The concrete forms of a task are either the Activity or the Sub-Process elements, which contain additional information modeling the actual service to be invoked as part of the task execution.

[2]The model could be extended in a similar way to add transactional properties to each task, as discussed in [98].

Attributes of the Process element:

Name (`Tag`)	Type	Description
Published (`PUBLISHED`)	Boolean	Whether the process should be published as a Web service by default. The published state of a process can be changed anytime, also after a process has been compiled and deployed.
SubProcess (`SUBPROC`)	Boolean	Whether the process can only be used as part of another process. If this flag is set, the user will not be able to start the process directly but only through other processes that invoke it. However, any process can be called from within any other process regardless of the value of this attribute.
Abstract (`ABSTRACT`)	Boolean	If this flag is set, the process is abstract: it is an empty process whose interface can be implemented by other processes. Apart from the interface, the rest of the content of an abstract process is ignored and will not be used to generate an executable process during compilation. Abstract processes can be used in conjunction with late binding[a], where only the interface of the Process to be called must be known at compile time.

Content of the Process element:

Name (`Tag`)	Element	Description
Views (`VIEWS`)	View	Container for the views over the data and control flow graphs of the process.
Dataflow (`DATAFLOW`)	Binding	Container for the global data flow graph of the process. The graph is stored as a set of bindings (directed edges) referencing pairs of parameters.
Constants (`CONSTS`)	Parameter	Container for the constant pool of the process. Constant values are modeled with Parameters, as this element already has all the attributes (Type and Value) required.
Tasks (`TASKS`)	Task	Container listing the tasks (Activites or SubProcesses) composing the process. A process which is not abstract cannot be empty, as it must contain at least one task.

[a] See Section 2 on page 36 for an example

Attributes of the Task element:

Name (`Tag`)	Type	Description
Activator (`ACT`)	String	The Activator is a boolean expression over the state of the tasks of the process, which determines when the task is ready to be activated. In most cases, the value for this attribute is automatically generated from the control flow graph, but it can also be changed by the user to specify more complex expressions.
Condition (`COND`)	String	The Condition is a boolean expression over the data flow of the process, which (for performance reasons) is evaluated only once when the task's activator fires. If both the condition and the activator are true, the task will be executed. Otherwise if the activator is true, but the condition is false, the state of the task will be set to unreachable.
Priority (`PRIORITY`)	Integer	Optional scheduling priority hint for the task. With this attribute, tasks found on the critical path of the process' control flow graph can get a priority boost.
Dependency (`DEP`)	DependencyType	Optional attribute used only for tasks part of a list-based loop, this specifies the type of dependency between the multiple instances of the task that are dynamically created for each list element.
Synchronization (`SYNCH`)	SynchType	Optional attribute used only if a task is part of a list-based loop with the Dependency attribute set to 'None'. This attribute specifies how to synchronize the various multiple instances running in parallel.
FailureHandling (`FAILH`)	FailureHandlingType	Optional attribute used in conjunction with parallel list-based loops. This attribute is used to control how failures of the multiple task instances affect the outcome of the list-based loop.

Values of the Dependency attribute of type DependencyType:

Value	Description
Finished	The tasks are executed sequentially, but only if the previous one in the sequence completed successfully.
Failed	Also sequential execution, but only if the previous task in the sequence has failed. This is useful for modeling a chain of alternative service invocations that should be tried only if a failure occurs.
FinishOrFailed	The tasks are executed sequentially, no matter what happens.
Aborted	Tasks are executed sequentially, but with the constraint that a task is started only if the previous one has been manually aborted by the user.
None	The multiple instances of the tasks are executed in parallel as there are no dependencies between them.

Values of the Synchronization attribute of type SynchType:

Value	Description
WaitForAll	Wait for all multiple instances to finish their parallel execution and merge the results of all instances.
WaitForOne	Finish as soon as one of the multiple instance finishes and ignore the rest, i.e. the results will not be merged.

Values of the FailureHandling attribute of type FailureHandlingType:

Value	Description
FailForOne	Fail as soon as one of the task instances has failed.
FailForAll	Fail only if all task instances have failed.
FailForPercent	Ignore up to a certain number of failed task instances.

The **Activity** (`ACTIVITY`) element extends the Task element – An Activity is a task which references a program.

Attributes of the Activity element:

Name (`Tag`)	Type	Description
ProgramID (`PROGRAMID`)	ref:Program	The `OID` of the program used to execute the activity.

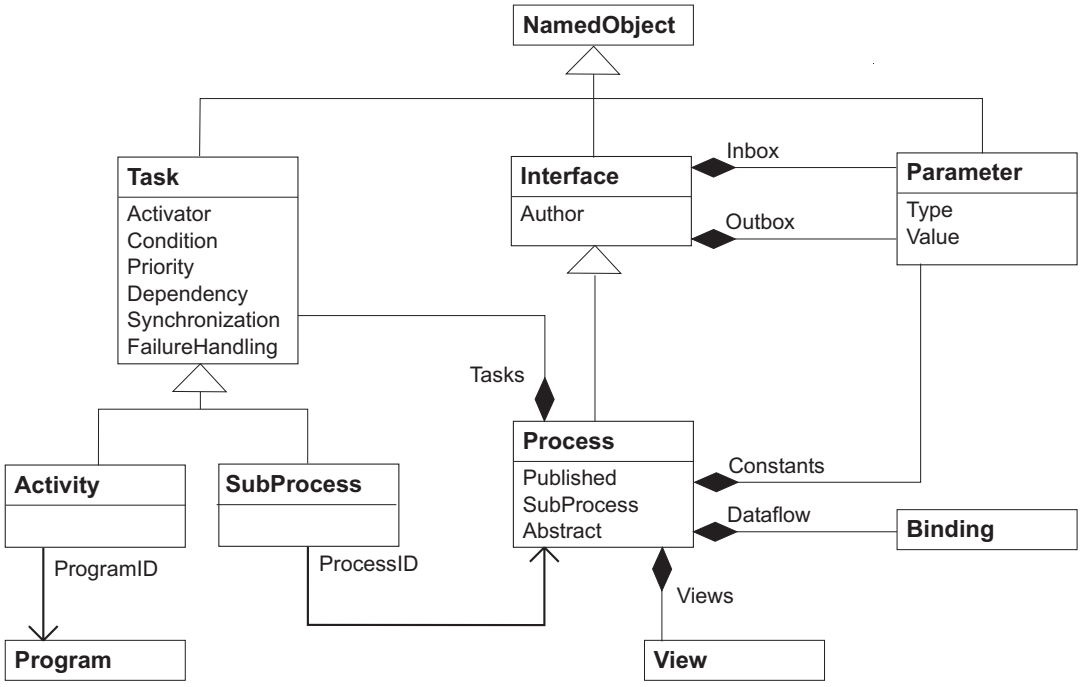


Figure 5.9.: *Structure of Processes and Tasks.*

The **SubProcess** (`SUBPROC`) element extends the Task element – A SubProcess is a task which references a process. In order to guarantee the extensibility of the model, the attributes which give more control on the process call (e.g. whether it should happen asynchronously) are not included in the SubProcess element. Instead, they are listed as system parameters, as specified in Section 4.8.2 on page 74.

Attributes of the SubProcess element:

Name (`Tag`)	Type	Description
ProcessID (`PROCESSID`)	ref:Process	The `OID` of the process which will be started while the sub-process is executed.

5.3.4. Data Flow Elements

The Parameter and Binding elements are used to model the data flow graph of a process (Figure 5.10). As previously mentioned, in most existing process modeling languages, the data flow between the various tasks of the process is left implicit [233]. In some languages, in order to connect two tasks and model the exchange of data between them, it is necessary to resort to global variables and assignment activities [29, 112]. In other cases, the input and output data exchanged by a task is modeled as a single (structured) parameter [144].

In our approach we found that a more detailed model of the data flow of a process, which includes multiple parameters for a task and avoids global variables, gives the following benefits:

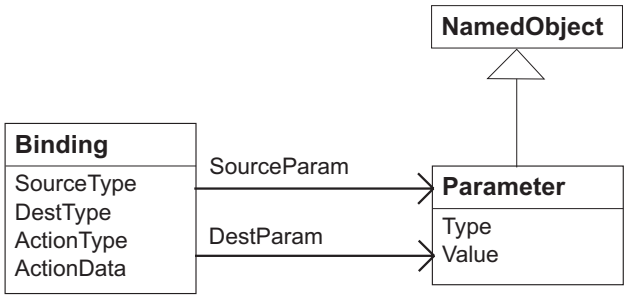


Figure 5.10.: *Structure of the data flow graph of a process.*

- *Abstraction.* As the data flow is modeled explicitly and visually as a graph where the bindings are drawn as arrows linking the source to the destination parameter, it is not necessary to manually schedule the data flow transfers between tasks by inserting assignment activities in the control flow of the process, so that the data flow transfers are carried out the correct time. A graph of the data flow is easier to program as it presents the developer with a higher level view over the data exchanges between the tasks of the process. If necessary, these can be automatically mapped to the correct assignment activities by a compiler. Thus, the possibility of making errors is reduced.

- *Clarity.* Global variables have been considered harmful for a long time [260] and, therefore, if they should be used sparingly in traditional programming languages, they are also not a good solution for meta-programming languages, such as service composition languages. One of the reasons is that the use of global variables adds unnecessary complexity to the model, as a data transfer between two service invocations must be modeled indirectly, i.e., through an intermediate global variable. A side-effects free model such as a data flow graph [104], where data is exchanged directly between the parameter of tasks, is much more clear to visualize, program, understand, debug and maintain.

- *Automatic control flow derivation.* If two tasks are linked with a data flow binding, this implies that there also is a control flow dependency, as the second task cannot start before its data is ready and data is ready when the first task has finished. Thus, it becomes possible to automatically derive the control flow dependencies between the tasks from the data flow graph.

- *Type checking.* If the data transfers between a set of service invocations must be explicitly modeled, type checking can be applied to ensure that only parameters with compatible data types are connected. Furthermore, the direct interconnection of incompatible service interfaces is detected and prevented. As we have shown in Example 4.3 on page 69, the data flow approach can also be useful to provide the necessary adapter and model the required transformation rules between mismatching data representations.

The **Parameter** (`PARAM`) element extends the NamedObject element – A Parameter models a data container that can be attached to a Process, a Program, an AccessMethod and a ComponentType.
An important decision taken during the design of the Opera Modeling Language was whether to include explicit data types or not. Given the benefits of static type checking, it would seem an easy choice. However, considering that the language is applied to the service composition domain, type safety may be quite difficult to achieve given the heterogeneity of the components involved. In fact, almost each different type of component that can be composed with OML comes with his own type system: Web services use XML Schema; Java components use the Java object oriented type system; UNIX applications interact at the level of textual data streams, which may be relatively unstructured and difficult to fit in a type system.
Although it may be difficult, but feasible to define and enforce mappings between different type systems [175], in JOpera we took the following approach. At compile-time, the optional type information associated with Parameters is used to warn the user about potential syntactical incompatibilities and conflicts. At runtime, the data to be produced and consumed by the services is stored within Parameters formatted as String. Where possible, this typeless solution (i.e., everything is a String) is complemented by applying the appropriate constraint checking operators to the data in transit. This way, at runtime, the consistency of the data can still be ensured.

Attributes of the Parameter element:

Name (`Tag`)	Type	Description
Type (`TYPE`)	String	Optional parameter type. Although all parameter values are stored internally using a string representation, types can be used to perform static type checking where data flow connections are allowed only between parameters of matching types.
Value (`VALUE`)	String	Optional default value. This is the initial value of the parameter, which may be overwritten during the execution of the process with data coming from other parameters or with the results returned by a task invocation.

The **Binding** (`BIND`) element extends the Object element – A data flow Binding is an edge linking a source and a destination parameter in the data flow graph of the Process. Parameters can belong to a process (including constants), or a task. For tasks, the parameters are defined in the referenced program/process. By default, a binding models a data transfer between a pair of parameters. Additionally, using the `ActionType` and `ActionData` attributes, it is possible to specify the application of a split or merge operator to the data in transit. As shown in Section 3.5 on page 31, this part of the model corresponds to the list-based loops, where the value of the source parameter is split into a list of values, and each of the values is assigned to a replica of the destination parameter. The ActionData attribute, in this case, encodes how the data is split (and later merged). The synchronization and failure handling of the multiple tasks are controlled by attributes stored within the task, as there can be multiple incoming (and outgoing) data flow bindings for the same task.

Attributes of the Binding element:

Name (`Tag`)	Type	Description
SourceType (`SOURCETYPE`)	BindRefType	The type of source container referenced by the `SOURCETID` attribute. It can be a normal parameter, a system parameter or a constant source.
SourceParam (`SOURCEPID`)	ref:Parameter	This attribute contains the `OID` of the source Parameter where the data is copied from.
DestType (`DESTTYPE`)	BindRefType	The type of Destination container referenced by the `DESTTID` attribute. It can be a normal parameter, a system parameter. It cannot be a constant, as they are immutable.
DestParam (`DESTPID`)	ref:Parameter	This attribute contains the `OID` of the destination Parameter and specifies where the data of the source parameter is written to.
ActionType (`ACTION`)	ActionType	Optional attribute, indicating the type of action to be performed when copying the data across the bound parameters..
ActionData (`ACTIONDATA`)	String	Optional attribute, with the parameters controlling the action to be performed. Currently it is used with the Split action to store the regular expression for splitting the incoming string and producing the resulting list of elements. By default the data is split based on a whitespace pattern. In the case of a Merge action, this attribute is used for storing how the elements of the list to be merged are concatenated together.

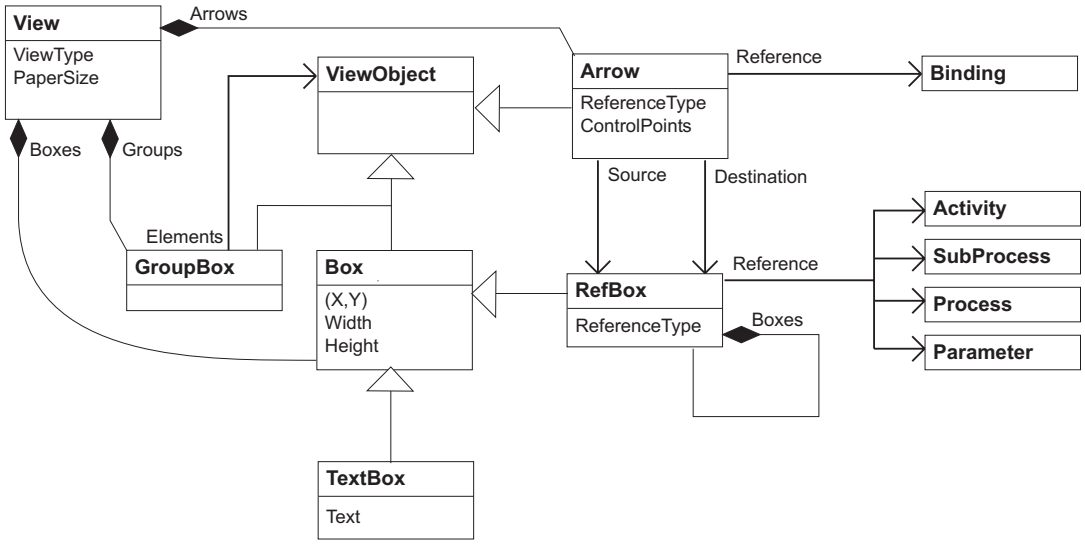


Figure 5.11.: *Structure of the elements of the JOpera Visual Composition Language.*

Values of the SourceType, DestType attributes of type BindRefType:

Value	Description
Normal	By default, the source (or destination) parameter referenced by the binding is a user-defined parameter.
System	The source (or destination) parameter is a system parameter.
Constant	The source parameter is a constant.

Values of the ActionType attribute of type ActionType:

Value	Description
Copy	The default binding action is to copy data across from the source parameter to the destination parameter.
Split	Split the content of the source parameter into a list and instantiate the task receiving the destination parameter for every element of the list.
Merge	Merge the multiple source parameters that have been created after a split into a single destination parameter.

5.3.5. JOpera Visual Composition Language (JVCL) Elements

This part of the meta-model (Figure 5.11) describes how the diagrams with the data and control flow graphs of a process (depicted using the JOpera Visual Composition Language) are serialized into an OML document. More precisely, there is a close relationship between the OML elements described in this section and their corresponding visual representation in JVCL. In practice, as shown in Figure 5.12, JVCL can be considered as the result of the graphical rendering of part of an OML document. The rendering operation is

controlled by the visual syntax rules defined in Chapter 3, which determine whether the graph stored in the OML document is displayed as a control flow or as a data flow graph. Furthermore, by selecting some additional options, the user may fine tune the appearance of the resulting diagram (e.g. parameter types may be shown or left hidden). To illustrate how the rendering of JVCL starting from an OML document works, Figure 5.13 shows an example. As it can be observed from the figure, the format, the size and the text displayed within each visual object of the JVCL language are controlled by the attributes of the corresponding OML elements..

The **View** (`VIEW`) element extends the NamedObject element – A view is a visual representation of the control flow or data flow graphs of the enclosing process element. A graph is modeled both in terms of its topology (edges linking nodes) as well as its two dimensional layout (nodes are displayed as rectangles located at certain coordinates, while edges can be routed following a set of control points).

Attributes of the View element:

Name (`Tag`)	Type	Description
ViewType (`VTYPE`)	ViewType	Indicates the type of view, determining the visual syntax to be used. Only one control flow view is allowed inside each process, but multiple, partial, overlapping views over the same data flow graph are allowed. If additional views over the tasks of a process become necessary (e.g. a transactional view), it would be possible to extend the process model by adding values to this attribute.
PaperSize (`PAPER`)	String	The default size of an empty view.

Content of the View element:

Name (`Tag`)	Element	Description
Arrows (`ARROWS`)	Arrow	Container for the edges (arrows) of the graph.
Boxes (`BOXES`)	Box	Container for the nodes (boxes) of the graph.
Groups (`GROUPS`)	GroupBox	Container for the groups of graph elements.

Values of the ViewType attribute of type ViewType:

Value	Description
Controlflow	The view uses the control flow syntax: boxes represent tasks and edges their control flow dependencies.
Dataflow	Views of this type conform to the data flow syntax: boxes represent both tasks and parameters, while edges represent data flow bindings.

The **ViewObject** element extends the Object element – Any element inside a View element should extend this one. In the case of Arrows and RefBoxes, it represents a visible object which references an element in the rest of the process model. For example: in order for a box to represent a task, the `OID` of the task visualized by the box is stored in the Reference attribute of the corresponding RefBox element. Considering that the same element can appear in multiple views, there is clear separation between the elements of the process model and their visual representation.

The **Arrow** (`ARROW`) element extends the ViewObject element – An arrow is a directed edge linking two boxes. Depending on the type of the container View element, it represents a control flow dependency between two task boxes, or a data flow binding between a pair of parameters. An arrow cannot exist without both of the boxes which it links.

Attributes of the Arrow element:

Name (`Tag`)	Type	Description
Source (`ID1`)	ref:RefBox	The source box, from which the arrow begins.
Destination (`ID2`)	ref:RefBox	The destination box, pointed to by the arrow.
ControlPoints (`CONTROLPOINTS`)	String	Optional list of control points coordinates. Although the simplest arrow directly connects the source with the destination box, it may be necessary to reroute the arrow through a set of intermediate points for improving the layout of the graph.
Reference (`REF`)	ref:Binding	Optional attribute used only for arrows of data flow views. In this case, this attribute contains the `OID` of the data flow binding element represented visually by the arrow.
ReferenceType (`REFTYPE`)	ARefType	This attribute controls the semantic of the arrow. Its values also depend on the type of the enclosing View. For arrows representing data flow bindings, this attributes caches the binding's `ActionType` attribute and is used to determine whether the split (or merge) icons should be displayed together with the arrow. In case of control flow dependencies, this attributes determines the actual type of dependency used, as discussed in Section 3.4 on page 25.

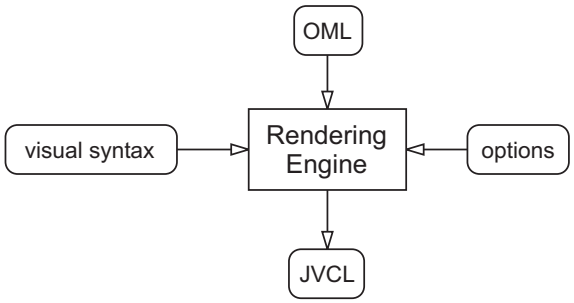


Figure 5.12.: *Relationship between the Opera Modeling Language and the JOpera Visual Composition Language.*

Values of the ReferenceType attribute of type ARefType:

Value	Description
Copy	(Dataflow) The default data flow arrow. It represents the copying of data from the source parameter box to the destination.
Split	(Dataflow) This arrow represents a split operation. The parameters controlling such operations are stored in the referenced binding.
Merge	(Dataflow) This arrow represents a merge operation.
Finished	(Controlflow) The default control flow arrow, it models the following dependency: the destination task is started only after the source task has finished.
Failed	(Controlflow) This arrow can be used for exception handling: the destination task is started only after the source task has finished.
FinishOrFail	(Controlflow) This arrow models a dependency independent on the outcome of the source task, as long as it was executed.
Aborted	(Controlflow) The destination task should be started only if the user aborts the source task.
Unreachable	(Controlflow) The destination task should be started only if the source task becomes unreachable.

The **GroupBox** (`GROUPBOX`) element extends the ViewObject element – A Group is a semantically transparent grouping of visual objects used to constrain the automatic layout algorithms.

Attributes of the GroupBox element:

Name (`Tag`)	Type	Description
Elements (`ELEMENTS`)	ref:ViewObject	List of the `OID` attributes of the grouped objects. It can also contain references to other groups, as long as no cycle is introduced.

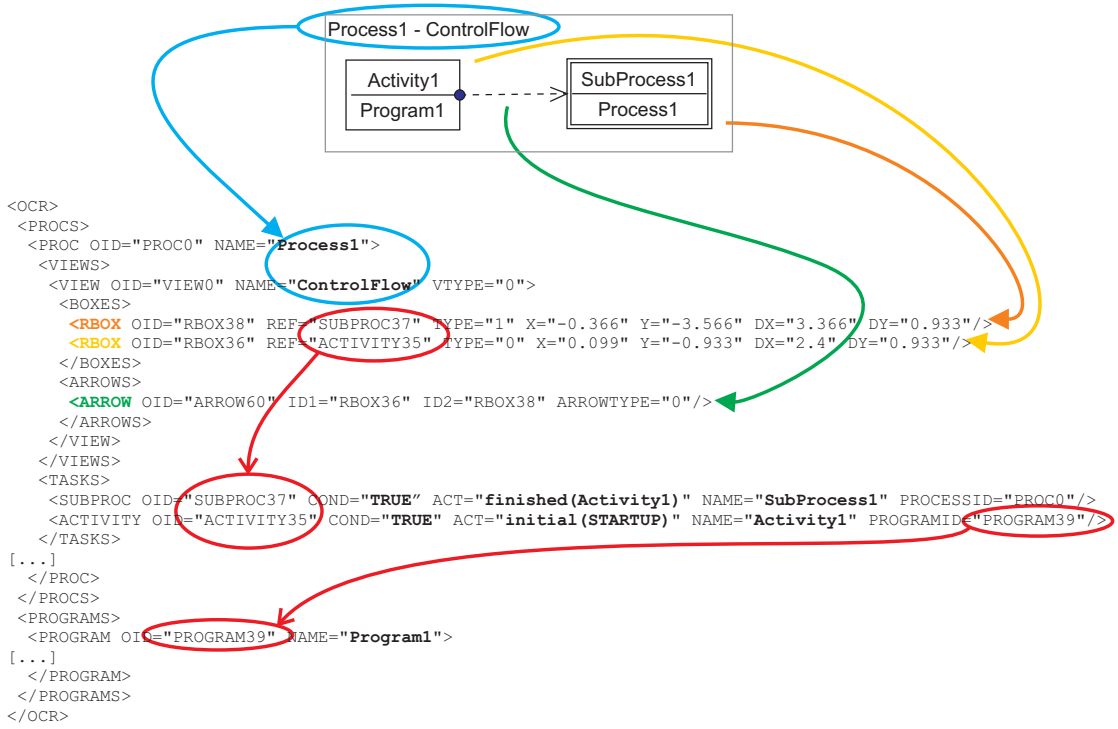


Figure 5.13.: *Example on how a JVCL control flow view is stored in the underlying OML model. Each visual object (the Process, the Activity, the Sub-Process and the Control flow dependency between them) corresponds to an element of the OML document. The text shown inside the boxes is stored in the corresponding document elements. The RefBox elements are linked to the referenced task elements, as it is possible to display the same task in multiple views.*

The **Box** element extends the ViewObject element – A box is any rectangle displayed in a view, it is further specialized by the TextBox and RefBox concrete elements. All coordinates of the graphic elements are stored using floating point values. Although the origin of the coordinate system should be mapped to the center of the screen, no explicit assumption is made about the direction of the X and Y axis.

Attributes of the Box element:

Name (`Tag`)	Type	Description
X (`X`)	Float	The X coordinate of the center of the box.
Y (`Y`)	Float	The Y coordinate of the center of the box.
Width (`DX`)	Float	The horizontal width of the box.
Height (`DY`)	Float	The vertical height of the box.

The **TextBox** (`TEXTBOX`) element extends the Box element – A TextBox is used to display textual comments inside a view. If a text box overlaps with any other ViewObjects, such objects appear as commented out and the corresponding process elements are ignored by the compiler.

Attributes of the TextBox element:

Name (`Tag`)	Type	Description
Text (`TEXT`)	String	The text of the comment entered by the user.

The **RefBox** (`RBOX`) element extends the Box element – A RefBox is a node of the graph, which represents a task, a parameter or the process itself. The text shown inside the RefBox is extracted from the referred document element. Depending on the configuration of the rendering engine, as shown in Figure 3.1, a task box may display both the name of the task and the name of its referenced program (or process). In case of data flow views, the box representing a parameter may show both its name and its type (or even part of its value, at runtime). These boxes are automatically resized to fit with the displayed text.

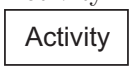
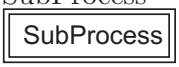
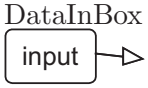
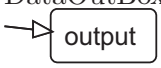
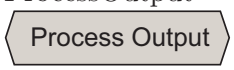
Attributes of the RefBox element:

Name (`Tag`)	Type	Description
Reference (`REF`)	ref:Task ref:Process ref:Parameter	Reference to the task, process or parameter element which is visually represented by this box element.
ReferenceType (`REFTYPE`)	BRefType	The type of the referenced object. This attribute controls the visual syntax used to display the box and is also used to indicate the expected location within the OML document of the referenced element.

Content of the RefBox element:

Name (`Tag`)	Element	Description
Boxes (`BOXES`)	RefBox	In case of data flow views, for efficiency reasons, the boxes representing parameters are stored nested inside the box representing the task (or the process) to which the parameters are attached to. This nesting is visually represented by drawing (implicit) arrows between the container box and its children, representing parameters. As defined in Section 3.3 on page 23, data flow parameters cannot be displayed without the task (or process) they belong to. Storing parameter boxes inside the container element has the additional benefit, that if the box representing a task is deleted, all of its children boxes representing the parameters are also deleted with it.

Values of the ReferenceType attribute of type BRefType:

Value	Description
Activity 	(Control and Dataflow) The element is a box which represents an Activity. The Reference attribute contains the `OID` of an Activity listed in the container process.
SubProcess 	(Control and Dataflow) The element is a box which represents a SubProcess. The Reference attribute contains the `OID` of a SubProcess of the container process.
DataInBox 	(Dataflow) These boxes can only be used as arrow destinations in data flow views. If a box representing an Activity/SubProcess contains a DataInBox, such DataInBox represents an *input* parameter of the Program/Process referenced by the task. As an exception, if a DataInbox is found inside a ProcessOutput box, then it represents the output parameter of the container process. The Reference attribute contains the `OID` of a Parameter.
DataOutBox 	(Dataflow) These boxes can only be used as arrow sources in data flow views. If a box representing an Activity/SubProcess contains a DataOutBox, such DataOutBox represents an *output* parameter of the Program/Process referenced by the task. As an exception, if a DataOutbox is found inside a ProcessInput box, then it represents the input parameter of the container process. The Reference attribute contains the `OID` of a Parameter.
ProcessInput 	(Dataflow) Placeholder for the input interface of the process. The input parameters of the process are stored inside this element. The Reference attribute contains the `OID` of the process element.
ProcessOutput 	(Dataflow) Placeholder representing the output interface of the process. The output parameters of the process are stored inside this element. The Reference attribute contains the `OID` of the process element.
Const	(Dataflow) This box represents a constant value. The Reference attribute contains the `OID` of a Parameter element inside the constant pool of the process. Unlike Parameter boxes, boxes representing constants are stored directly inside a View element.
SysInBox 	(Dataflow) This box represents a system input parameter, which can be used only as destination of data flow arrows. The Reference attribute contains the `OID` of a Parameter element inside the input parameters of an AccessMethod element.
SysOutBox 	(Dataflow) This box represents a system output parameter, which can be used only as source of data flow arrows. The Reference attribute contains the `OID` of a Parameter element inside the output parameters of an AccessMethod element.

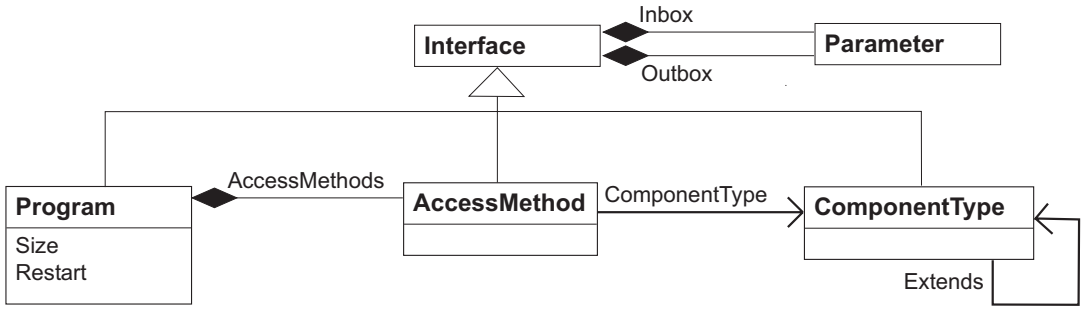


Figure 5.14.: *Structure of the Program and Component type library.*

5.3.6. Program Library Elements

These elements are used to model the program library (Figure 5.14). By design, in the Opera Modeling Language there is a clear separation between the processes, which define the composition, from the programs, which define the components. Thus, a library of reusable program definitions can be built and shared among multiple processes composing them in different ways.

The **Program** (`PROGRAM`) element extends the Interface element – A Program is a component referred by the activities of a process and represents the invocation of a service belonging to one of the types described in Chapter 4. Program definitions can be entered manually or automatically imported from other Interface Description Languages, such as WSDL. Similar to WSDL, a Program both includes information on the interface of a service, as well as multiple AccessMethod elements which refer to the actual component type to be used while accessing the service..

Attributes of the Program element:

Name (`Tag`)	Type	Description
Size (`SIZE`)	Integer	Scheduling hint to classify the program in terms of its execution cost (expected duration and resource requirements). The scheduler may use this hint to implement a smallest-job-first heuristic [133].
Restart (`RESTART`)	Integer	Maximum number of automatic restarts on failure (-1 means unlimited). In case of a Program containing multiple AccessMethods, the invocation of the service will be retried using a different one each time.

Content of the Program element:

Name (`Tag`)	Element	Description
Access Methods (`ACCESS`)	AccessMethod	In order to be executable, a Program must contain at least one access method. Multiple access methods are treated as alternative, equivalent paths to invoke the same service.

The **AccessMethod** (`METHOD`) element extends the Interface element – An access method contains the system parameters that control how to access the services provided by the specified component type. It also defines a mapping between the program input parameters – which are application dependent and defined by the user – to the system input parameters of the ComponentType, which depend on the mechanisms and the protocols used to perform the service invocation.

Attributes of the AccessMethod element:

Name (`Tag`)	Type	Description
ComponentType (`COMP`)	ref:ComponentType	The type of component that should be used as a template to define the set of system parameters of this access method.

5.3.7. Component Type Modeling Elements

This part of an OML document describes the component types that are used as templates for the program's access methods and corresponds to the description of the various types of components that can be used with JOpera presented in Chapter 4. In practice, this part of the model is defined in a separate document, which is part of the JOpera configuration and should be included by the program definitions used by the processes. The set of available component types should correspond to the configuration of the JOpera kernel. At runtime, in order to perform the service invocation, the set of system parameters of each component type is passed to the corresponding service invocation adapter as described in Section 7.4 on page 163.

The **ComponentType** (`COMP`) element extends the Interface element – A Component Type is a template for an AccessMethod, it defines a set of input and output system parameters that are used to generate the ones of the access method referencing it.

Attributes of the ComponentType element:

Name (`Tag`)	Type	Description
Extends (`PARENT`)	ref:ComponentType	Refers to another component type which is extended by the current one. All of the system parameters defined in the parent component type are inherited by the current one.
Abstract (`ABSTRACT`)	Boolean	This flag indicates whether the component type can be directly used as a template for generating Access Methods.

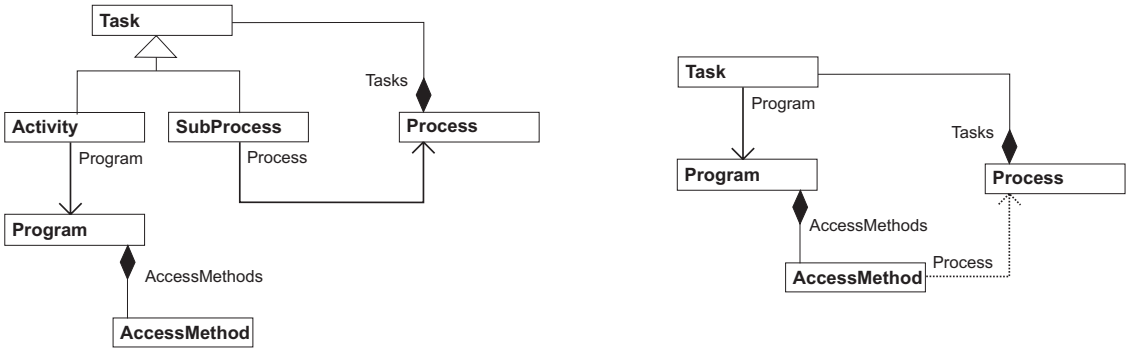


Figure 5.15.: *Simplified Task Model. Comparison between the original OML model (left), which includes the Activity and SubProcess elements, with the simplified model (right), where there are only Tasks.*

5.4. Discussion

Although the description of the Opera Modeling Language in this chapter presents 20 different elements (including the document root element), we have tried to keep OML as simple as possible[3]. First of all, as it can be seen from Figure 5.4, six elements are abstract: Object, NamedObject, Interface, Task, ViewObject, and Box. They are very useful in shortening the description by factoring out commonalities, but they will never appear in a concrete OML document. Furthermore, within the visual elements, the TextBox and GroupBox are used to provide convenience features such as layout constraints and visual comments, but are not strictly necessary to store the information defining the process model. Out of the remaining 11 elements, forming the core of the language, we distinguish two groups. On the one hand, the Process, Activity and SubProcess elements model the content of a process, while its structure in terms of data flow and control flow is modeled by the Parameter, Binding, View, Box and Arrow elements. On the other hand, the remaining three OML elements (Program, AccessMethod, and ComponentType) are not part of the process model itself, but form the model of the services that can be used as process components.

In this discussion, also with the aim of keeping the process model as simple as possible, the option of having Activity and SubProcess as separate elements should be compared with a smaller model where only Tasks are used to represent both constructs (Figure 5.15). As we already anticipated in Section 4.8.2 on page 74, during compilation SubProcess elements are mapped to a special type of component, modeling the internal invocation of a Process within the JOpera system. Therefore, instead of hiding such transformation inside the compiler, it could be possible to make it explicit in the original process model. Thus, the different Activity and SubProcess elements would blend into a single element called Task, which would reference a Program, and in the case of SubProcesses it would contain one AccessMethod of

[3]Other process modeling languages are much more complex: BPEL4WS [112], for example, has 44 elements, BPML [29] has 47.

type "Process Invocation", with system parameters matching the attributes of the original SubProcess element, and referencing – as indicated by the dotted arrow in Figure 5.15 (right) – the Process to be called.

With this approach, the model would be slightly simpler, and closer to what the compiler does in terms of preparing a process for execution. However, saving a transformation at compilation time will require additional complexity at design time, as we believe that – from the user perspective – the distinction between Activity and SubProcess remains useful. As first discussed in [21], it is important to make, at least at the syntactical level, a difference between basic and complex tasks. This way, by looking at the visual syntax of tasks within a control and data flow graph (Figure 3.1), the developer may easily distinguish Activities, representing an atomic step in the process execution, from SubProcesses, grouping together multiple steps.

Another important point should be mentioned regarding the way OML is intended to be used to program process-based service composition. Although OML defines the information that needs to be provided in order to define a proces, thanks to the JOpera visual development environment, the XML-based syntax of an OML document remains hidden from the developer at all times, as processes are built by literally drawing them, according to the syntax of the JOpera Visual Composition Language.

This approach explains the presence of some of the features of the OML metamodel, such as the heavy use of key-based references between elements and the storage of model checking status information within a process model itself. On the one hand, these features make it quite difficult to write an OML document manually, i.e., by entering XML code directly. On the other hand, OML leverages an XML-based syntax and contains the necessary features to facilitate the building of the appropriate editing and validation tools, so that the processes can be developed visually, at a higher level of abstraction.

Part II.

The JOpera System

6. Compiler

This chapter, in which we introduce JOpera's compiler, links the description of the language to the description of the system for executing it. Before we give a detailed presentation of JOpera's architecture (Chapter 7), which provides a flexible platform for the execution of the processes modeled with the Opera Modeling Language, in this chapter we describe how to transform (or compile) such process models into other representations, which are more suitable for execution.

6.1. Motivation

Once a process has been defined using the JOpera Visual Composition Language (Chapter 3) based on the underlying Opera Modeling Language (Chapter 5), there are two main alternative options to be considered in order to achieve its automatic execution by a process management system, as depicted in Figure 6.1:

- One solution is to directly interpret a process description. In the simplest case, the process description as it is specified by the user is directly executed by a process engine, which uses it both to generate new process instances and to interpret the associated state information (usually stored in a database) in order to invoke the various tasks of the process in the correct order.

- A similar solution – followed by most existing systems (e.g. [83, 92, 98, 124, 168, 201, 253]) – involves the transformation of the process description into an intermediate representation, e.g., in form of a database schema. Using an intermediate representation has the advantage that process descriptions can be stored in a form optimized for interpreted execution, which is still independent of the underlying operating system/hardware platform. Furthermore, if the intermediate representation is stored in a database, the generation of new process instances can be implemented using internal mechanisms of the database.

- Another solution (and one of the main contributions of this dissertation) consists of the idea that process descriptions produced by the user can (and should) be compiled into *executable code* to achieve a more efficient execution. By using Java as a target language and platform, such code still maintains useful portability properties. Additionally, it is possible to repurpose a Java virtual machine as a "process interpreter", as we will present in Section 6.5.

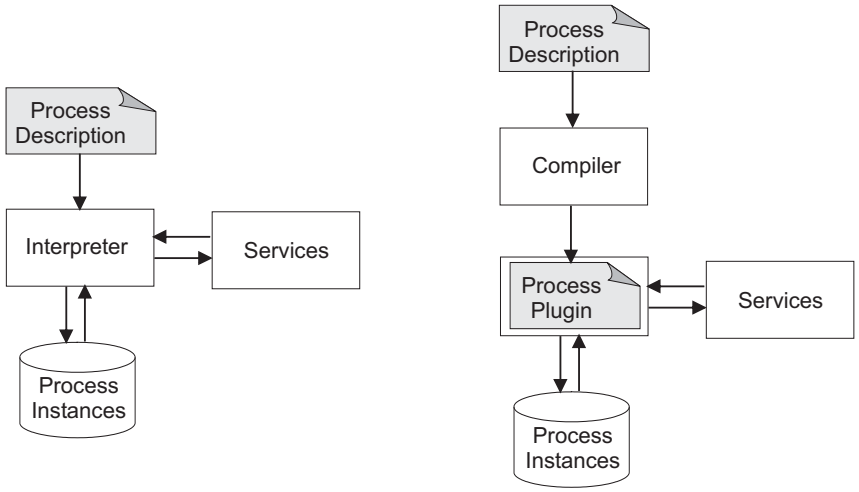


Figure 6.1.: *Alternative approaches to process execution.*
In order to execute them, process descriptions can be either interpreted (left) or compiled (right).

In JOpera, the information defining the processes is structured in a way to emphasize its understandability by human developers (JVCL) and ease of editing with automated tools (OML), while leaving concerns about efficient execution in the background. To illustrate this, as an example among many possible ones, we consider the data flow graph of a process, which is stored as a list of edges linking input and output parameters in an OML document[1]. This approach makes it easy to edit the graph by adding and removing individual data flow connections. However, at runtime, when the data flow transfers need to be carried out, it is more efficient to represent the same data flow graph as a set of assignment rules, grouping together the data flow connections that have to be triggered before the invocation of each task. As it can be seen from this simple example, it is quite difficult to define a generally optimal process model. Thus, in our approach we have chosen to use several models, each optimized for a different goal. More specifically, we distinguish between a design-time model, and two run-time models, one targeting process execution and the other intended for process monitoring purposes.

To go from the design-time model to the corresponding run-time representation, a process undergoes several transformations in order to prepare it for efficient execution. We refer to these transformations as a whole with the term *compilation*. Similar to traditional programming languages where the original source code is compiled into executable object code for a target platform, also the JOpera Visual Composition Language can be mapped onto various executable representations suitable for several different process execution engines. In this chapter we describe the main properties of a compiler for the following three execution environments:

[1]See Section 5.3.4 on page 101 for more information on the structure of a OML document, the underlying data representation of the JOpera Visual Composition Language

1. When targeting the BioOpera process support system [21], the required OCR textual representation of the processes is automatically generated from the OML model. As BioOpera was the original target execution environment for the Opera Modeling Language, the OCR and OML models share most of the basic concepts and assumptions, therefore the mapping between the two representations is quite straightforward.

2. Similarly, under some conditions, a representation in the form of the Business Process Execution Language for Web Services (BPEL4WS [112]) can be automatically produced. In order to achieve full compatibility, the JOpera Visual Composition Language processes should be built out of a restricted set of component types.

3. The third execution environment for which we have designed a compiler is the JOpera system itself. This particular compiler does not produce as output another executable process model to be instantiated and interpreted by the execution environment. Instead, the processes written in JOpera Visual Composition Language are compiled into Java executable code which is then compiled one more time so that the resulting bytecode is dynamically loaded into JOpera's runtime kernel for managing the concurrent execution of multiple process instances. Thanks to this approach, the internal structure of the JOpera's kernel is greatly simplified, as most of the complexity required at runtime to interpret the process models is shifted into the compiler, which can perform several optimizations and leverage many features of the existing Java language compiler.

6.2. Compiler's Architecture

Before presenting the most important aspects of the mapping of OML processes to OCR, BPEL and Java, in this section we give a brief overview over the compiler's architecture, as it is shared by all mappings. This way, we explain how the mappings can be applied to the original model to transform it into an executable representation for a given runtime platform.

The compiler has a layered architecture (Figure 6.2), where different parts of the process model are analyzed separately. The results of the analysis are then merged by the code emitter to produce the process representation in the target language (Java, BPEL or OCR). As in most modern compilers [50], a multi stage approach allows to separate model checking and analysis features from the back-end layer, which is responsible for generating the actual code in the target language.

Starting from the process model which contains the control flow and data flow graphs as well as the descriptions of the required components, the analysis layer uses different intermediate representations:

- For the control flow graph, a set of ECA rules is produced to describe the dependencies associated to each task. In Java, these are mapped to conditional

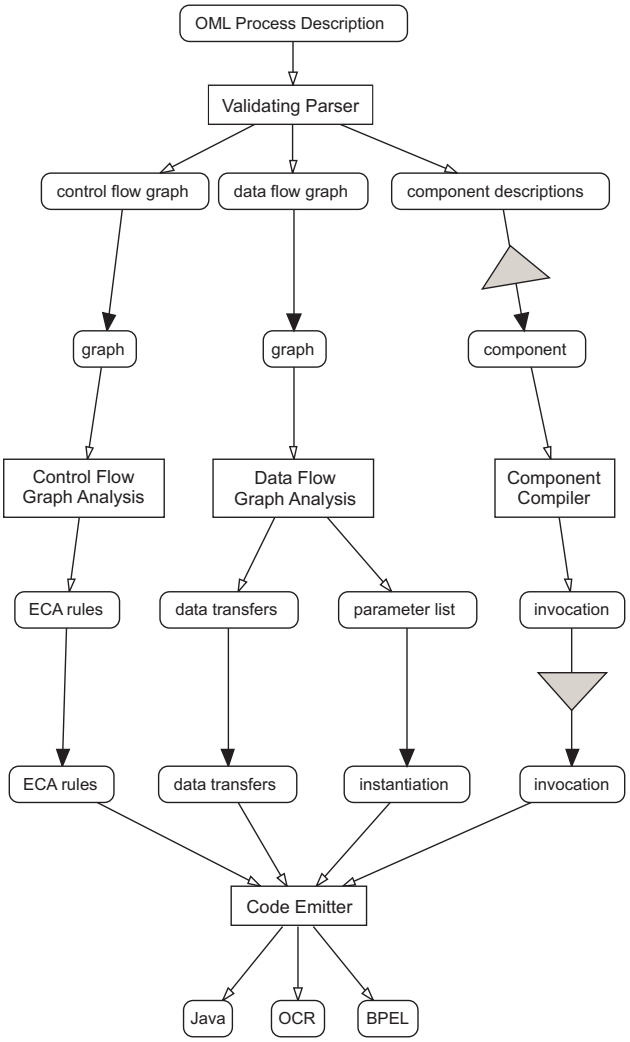


Figure 6.2.: *Multi-stage architecture of the OML compiler*

statements within the navigation code. In OCR and BPEL these rules are used to produce the correct dependencies and conditions associated with the tasks.

- The data flow graph analyzer extracts a schedule for the data transfers that should be executed before and after the invocation of each task. In Java, this is used to produce the corresponding data transfer code. In BPEL, these data transfers are mapped to *assign* activities, while in OCR the incoming and outgoing bindings are associated to each task[2].

  Furthermore, the data flow analysis also produces a set of parameter lists, which are used to shape the image of the process instance at runtime. In Java, this amounts to generating the corresponding instantiation code. In BPEL, this information can be used to produce the list of global variables, while in

[2]See Section 6.3.1 on page 123 for a description of the algorithm used to do this.

the OCR emitter it is used to define the input/output interfaces of processes and programs.

- The component descriptions are translated to the code for setting up the actual service invocation. This gives a certain flexibility in providing support for different component types. Depending on the target language (e.g., BPEL vs. OCR), not all component types may be supported. Nevertheless, the code emitter can check this and ensure that only the appropriate component types are used. Depending on the component type, some optimizations can be performed in order to leverage the specific characteristics of the component type. In the case of Java scripts[3], these can directly be embedded within the generated Java code.

Finally, the code emitter gathers the various parts of the intermediate representation of the process and uses it to produce the corresponding OCR, BPEL, and Java code, as we will discuss in the rest of the chapter.

6.3. Mapping to OCR

The Opera Canonical Representation (OCR) is the process modeling language first defined in [98] with the intent of providing a "common assembly language" for process modeling. By design, it contains features shared among many process modeling languages in order to facilitate their mapping onto it. Extended with resource modeling features, OCR is also the process modeling language for the BioOpera system [21] for which the first version of the JVCL/OML compiler was developed.

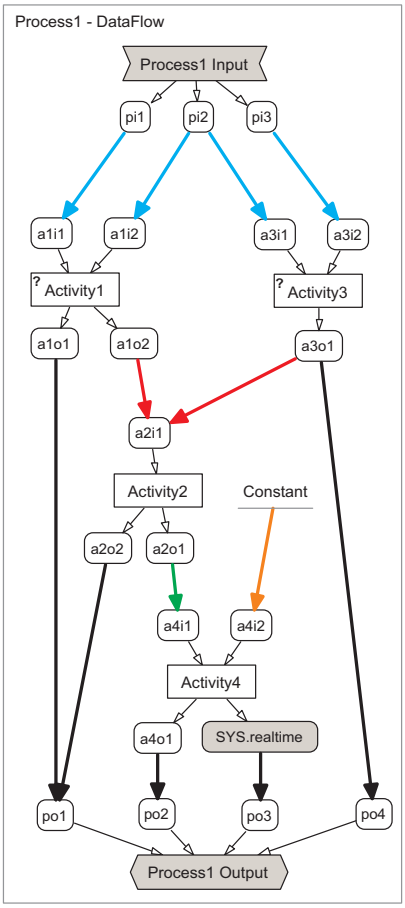
Given the very close ties between the OCR and OML process models, most of the compilation amounts to translating the process model from XML to OCR's textual syntax. For most of the OML elements, there is a similar OCR tag describing the processes and their component tasks, as well as the programs[4]. However, OML is a superset of OCR as it includes the serialization of the JVCL visual process definition. This part of the OML model is lost when performing the translation to OCR, which can thus be considered a one-way transformation between the two process models. Furthermore, there are also some differences regarding the structure of the program library and the representation of the data flow, which we will cover in the rest of this section.

6.3.1. Data flow mapping

In order to produce a suitable OCR representation, the data flow graph of an OML process must be analyzed and transformed according to the following algorithm. An example of the results of the algorithm applied to a simple data flow graph are shown in Figure 6.3.

[3]This particular type of component is described in Section 4.5.1 on page 60.

[4]For more information on the actual OCR syntax we refer the reader to the original definition in [21] and [98].




```
PROCESS "Process1" ( "pi1", "pi2", "pi3" )
      RETURNS ( "po1", "po2", "po3", "po4" )
      WHITEBOARD ( "Activity2_a2i1", "pi1", "pi2", "pi3",
                                     "po1", "po2", "po3", "po4" ),
      RESTART NOT-RESTARTABLE
TASKS
      PROGRAM "Activity1" : "_Program1"
            ( a1i1 = PROC.pi1, a1i2 = PROC.pi2 )
            STORE ( a1o2 > Activity2_a2i1, a1o1 > po1 )
      (
            ACT initial(STARTUP),
            COND PROC.pi2 < 0
      );
      PROGRAM "Activity2" : "_Program2"
            ( a2i1 = WB.Activity2_a2i1 )
            STORE ( a2o2 > po1 )
      (
            ACT finished(Activity1) and finished(Activity3),
            COND TRUE
      );
      PROGRAM "Activity3" : "_Program3"
            ( a3i1 = PROC.pi2, a3i2 = PROC.pi3 )
            STORE ( a3o1 > Activity2_a2i1, a3o1 > po4 )
      (
            ACT initial(STARTUP),
            COND PROC.pi2 > 0
      );
      PROGRAM "Activity4" : "_Program4"
            ( a4i1 = Activity2.a2o1, a4i2 = "Constant" )
            STORE ( a4o1 > po2, SYS.realtime > po3 )
      (
            ACT finished(Activity2),
            COND TRUE
      );

END TASKS
META
      NOTEXEC
      AUTHOR "Cesare Pautasso"
END META
END PROCESS
```

Figure 6.3.: *Example showing a mapping from the JVCL visual syntax to the OCR textual syntax of a simple data flow graph of a process. (The intermediate OML representation is not shown)*

1. First of all, the original OML data flow bindings are grouped according to the task to which the binding's destination parameter belongs to. Since in OCR the source of each input parameter of a task must be specified together with the invocation of such task, this grouping allows the compiler to generate such OCR binding lists for each task.

2. Then, some special cases must be addressed, as it is possible to have input parameters with multiple incoming bindings that link them to different source parameters. In the OCR model only one source for each input binding is allowed, therefore it is necessary to resort to the indirection provided by the whiteboard, which contains a set of global parameters, visible only inside a process.

   a) For each of the input parameters having multiple incoming bindings a whiteboard parameter is created. In order to guarantee its uniqueness,

this whiteboard parameter is named by appending the input parameter's name to the name of its container task.

b) For each of the incoming bindings, an outgoing binding from the source parameter to the new whiteboard parameter is created. Each of these outgoing bindings is associated with the task that owns the source output parameter.

c) An incoming binding between the whiteboard parameter and the original input parameter is added.

3. Another special case concerns bindings connecting output parameters of tasks to the output parameters of processes. Again, in OCR it is not possible to model this directly. Instead, the whiteboard is also used to shadow the process output parameters, i.e., when a process finishes, the whiteboard parameters are copied into the matching process output parameters. Therefore:

   a) For each process output parameter, a matching whiteboard parameter is created.

   b) For each binding to a process output parameter, an outgoing binding is added to the task that owns the source parameter linking it to the whiteboard parameter corresponding to the destination parameter in the process output.

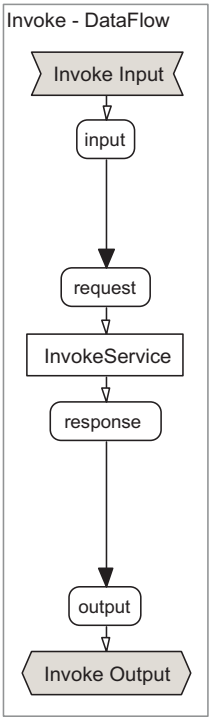
4. Finally, for each incoming binding of each task, depending on the type of the source parameter of the OML binding, the appropriate prefix must be appended to the OCR parameter. Such prefix identifies whether the source parameter of the binding belongs to the process (`PROC`), to a system parameter (`SYS`) or to the whiteboard (`WB`). If the source parameter is an output parameter of a task, then it is prefixed with the task's name.

If the JVCL data flow graph contains bindings associated with split or merge operators, the mapping becomes more complex, as OCR does not support such operators natively.

6.4. BPEL Mapping

In this section we show to what extent it is possible to map our visual composition language to the Business Process Execution Language for Web Services (BPEL [112]) version 1.1[5], an emerging XML-based specification for Web service composition, and viceversa. The main goal of such mapping is to be able to use the JOpera platform for visually composing Web services into processes, which can be later translated into BPEL (or any other equivalent specification) for external execution. Conversely,

[5]This specification is currently undergoing a standardization process and, also taking into account the volatility of Web service related standards, some of the details we take for granted in this section may become obsolete in time



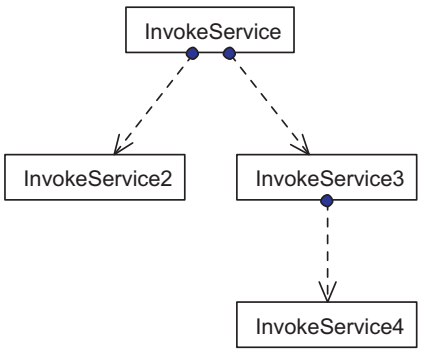

```
<process name="Invoke">
  <variables>
    <variable name="input"/>
    <variable name="output"/>
    <variable name="request"/>
    <variable name="response"/>
  </variables>

  <sequence>
    <receive name="receiveInput"
             portType="tns:Invoke"
             operation="initiate"
             variable="input"
             createInstance="yes">
    </receive>
    <assign>
      <copy>
        <from variable="input" part="parameters"
              query="//value"/>
        <to variable="request" part="parameters"
            query="/Service/value"/>
      </copy>
    </assign>
    <invoke name="InvokeService"
            portType="tns:Service"
            operation="Service"
            inputVariable="request"
            outputVariable="response">
    </invoke>
    <assign>
      <copy>
        <from variable="response" part="parameters"
              query="//result"/>
        <to variable="output" part="parameters"
            query="/onInvokeResult/result"/>
      </copy>
    </assign>
    <invoke name="replyOutput"
            portType="tns:InvokeCallback"
            operation="onResult"
            inputVariable="output">
    </invoke>
  </sequence>
</process>
```

Figure 6.4.: *A process with a single Web service invocation represented both in JVCL and in BPEL. The corresponding parts of the process are shown side by side.*

a BPEL document can be translated into OML when it is imported into JOpera to take advantage of its scalable execution facilities and visual monitoring environment.

6.4.1. Mapping to BPEL

The components of a JVCL process can be accessed using various mechanisms, which are not limited to those compatible with SOAP/WSDL. In order to keep the mapping feasible we will assume that either all tasks of a process represent Web service invocations or that Web service wrappers for the other classes of components can be readily provided. Such wrapping could be done automatically, as part of this mapping procedure, or manually, in a separate step.




```
<sequence>
  <invoke name="InvokeService”.../>
  <flow>
     <sequence>
       <invoke name="InvokeService2”.../>
     </sequence>
     <sequence>
       <invoke name="InvokeService3”.../>
       <invoke name="InvokeService4”.../>
     </sequence>
   </flow>
</sequence>
```

Figure 6.5.: *A process invoking multiple Web services shown both in JVCL and in BPEL. There is a clear correspondence between the block-based structure of the BPEL representation and the control flow graph of the JVCL.*

Partners For each of the JVCL activities invoking a Web service, a BPEL *partner* is created which contains a service link corresponding to the activity's program and also a BPEL *invoke* activity is prepared. For each of the JVCL sub-processes a *link* to the JOpera systems where the process is accessible, or, alternatively, to make the final BPEL document self contained, a new *scope* is added to the BPEL *process*.

Control Flow In general, given the arbitrary topology of control flow connections in JVCL it will not always be possible to reduce it to the block-structured control flow description of BPEL. However, the control flow graph of a JVCL process can always be mapped to a single BPEL *flow* activity composed of all the tasks of the process, with a direct translation of the dependencies between the service invocations. In case of control flow dependencies used to model exception handling, specific BPEL constructs can be employed. In case of loops in the JVCL control flow graph, they can be detected and mapped to a BPEL *while* block.

Data Flow In BPEL the flow of information between the services is not explicitly modeled with a data flow graph as in JVCL. Instead global variables (or containers, depending on the version of the specification) are used as temporary storage for the messages exchanged by the services and XPath expressions are used to refer to individual data elements of the messages. To map the data flow graph of a JVCL process, a BPEL *variable* to store the request/response messages of each service is created and for each data flow connections in the JVCL graph a BPEL *assign* activity is inserted before and after the service invocation represented by the BPEL *invoke* activity. This assignment activity contains the XPath expression used to access the individual JVCL parameter (or message parts). As an alternative, a BPEL *variable* for each JVCL data flow parameter can be added. An example of a basic data flow mapping is shown in Figure 6.4.

There are a few constructs in JVCL that have no equivalent construct in BPEL.

In addition to explicit control flow loops, JVCL offers iteration through list based split/merge data flow operators. It is unclear how this could be mapped to an existing standard BPEL construct. Furthermore the system parameters of JVCL, which, for example give access to meta-data about the process and its tasks, and can be used to specify the late binding of a service interface to its implementation, cannot be mapped to standard BPEL expressions.

6.4.2. Mapping from BPEL

BPEL mixes two different types of elements in the same process modeling language. On the one hand there are structural constructs for modeling the control flow and data flow of a process. On the other hand there are several different basic activities used to model the synchronous and asynchronous invocation of Web services, as well as the handling of events and alarms. The first type of elements can be mapped to constructs of the JVCL language, while the second type cannot be mapped directly but is implemented using a library of BPEL components[6].

Control Flow As the BPEL control flow is expressed using both a block and graph structure, it is always possible to map this to a pure graph based model (Figure 6.5). This way, BPEL constructs such as *sequence*, *flow*, *pick*, *while* and *switch* can be replaced by a corresponding combination of control flow dependencies and conditions.

BPEL structured exception handling (based on *throw*, *catch*, *catch all* activities) can be mapped to JVCL by introducing rule-based exception handlers and an ad-hoc component in the BPEL library which always fails and is used to represent the *throw* activity.

Data Flow The mapping of the data flow of a process is not so straightforward, given the use of global variables (or containers) and arbitrary *assign* activities in BPEL. To do so there are two main possibilities: either an *assign* activity can be mapped to a direct data flow connection between a pair of JVCL parameters or it is necessary to add an explicit task, which runs the XPath expression contained in the *assign* activity and transforms the input into the output parameter accordingly.

Messaging BPEL *invoke* activities can be directly mapped to JVCL activities with a reference to a program representing the corresponding service invocation. Furthermore, mapping BPEL activities such as *send*, *receive*, and *wait* can all be done by using JVCL tasks of the Messaging component library[7].

Events A similar approach is used with the *onAlarm/onMessage* constructs. In this case a process (or part of a process) could be set up as follows. There is a task which corresponds to the *onAlarm/onMessage*. This task terminates its execution

[6] See Section 4.11 on page 83 for more information about the components which represent some of the basic BPEL activities

[7] See Section 4.10 on page 79 for a description of the messaging components.

on receipt of a message, or on occurrence of an alarm. The block of actions to be carried out when such an event occurs, is translated as before with the additional dependency from the task corresponding to the original *onAlarm/onMessage*.

6.5. Mapping to Java

In this section we present how the JOpera Visual Composition Language is compiled into Java executable code. To do so, we first give some background on process navigation and instantiation. For a given process, these two procedures are executed by invoking the Java code produced by the compiler. Then we introduce the finite state machine that models the possible execution states of processes and their tasks. Understanding this finite state machine is also important because it is the basis for the interaction between task state changes and their control flow dependencies, as it is performed by the process navigation algorithm. As a concluding example, we show how the compiler uses all of these aspects to generate the executable Java code for a process with a simple structure.

6.5.1. Process Navigation

Navigation is the procedure whereby the system determines the set of tasks to be executed next [124], given the current state of the process and its control flow graph, specifying the partial order of execution of the tasks. To do so, the navigation procedure interprets the information of a directed graph, where the nodes represent the tasks and are labeled with their current state, and the edges represent control flow dependencies between the tasks. When a state change of a task occurs, the algorithm proceeds in two steps. First, in order to determine the set of tasks affected by the state change, it follows all outgoing control flow dependencies. Then, it evaluates the starting conditions of these tasks to check if they are ready to be started. This way, after every task state change it is possible to determine the set of tasks to be started next.

This approach is very similar to mapping the process description to a set of Event, Condition, Action (ECA) rules. State changes of tasks trigger events, which will cause the evaluation of the conditions associated with the set of dependent tasks and, when a condition evaluates to true and one of these rules fires, the actions required to start the tasks can be carried out. More specifically, during navigation the system mainly performs two types of actions (Figure 6.6):

1. The first type concerns the actual task execution, i.e., packing all the necessary information into a job that can be submitted to the scheduler responsible for finding a suitable provider for invoking the service.

2. The second type groups operations that access or modify the state information of a process. For example, copying the data from the parameters of one task to another as specified in the data flow graph of the process, as well as setting metadata values, such as the starting time of a task, or accessing the state of

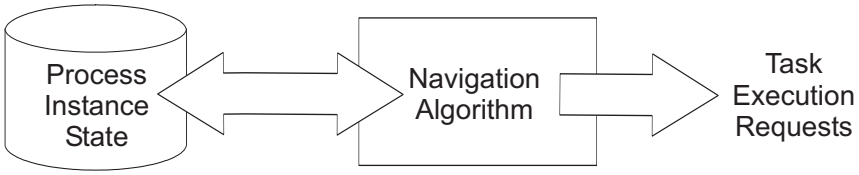


Figure 6.6.: *Process navigation actions*

a set of tasks to determine whether they have failed in order to trigger the corresponding exception handler.

Executing Navigation In practice, it is not necessary to compile a process description into a generic ECA-like representation to be interpreted by the process engine. Instead, to implement our process navigation algorithm we build on the idea of mapping the process description to a program embodying the specific rules corresponding to the process description, generated using an ordinary programming language. This way, we can use the language's compiler to produce executable code which can be then dynamically loaded and linked into the kernel's runtime environment to be executed. This approach has the potential to provide better performance. First of all, the executable code is generated in a standard programming language, in our case Java[8], which then is compiled one more time. This way we can map the process structure to standard language constructs, which can be efficiently executed. Moreover, during the generation of the code it is possible to analyze the structure of the process and perform optimizations.

In addition to this, the generated program is completely stateless, as it only contains a mapping of the process structure. To perform navigation over a particular process instance, the program reads its state as input. Therefore, it is possible to perform navigation over many instances of the same process using the same program code, which only needs to be loaded once. In many existing systems, this clear separation between the state of a process instance and its structure is missing and, in the worst case, both types of information need to be loaded from the persistent repository before each invocation of the navigation procedure, incurring in unnecessary overhead.

6.5.2. Process Instantiation

Process instantiation is the procedure whereby the system creates a new *instance* of a given process *template* (description, or type) so that it can be executed with the navigation algorithm presented in the previous section. The input to the process instantiation procedure consists of the name of the process template that should be

[8]In our prototype we choose Java because of its portability, the maturity of the available tools and its ease in dealing with dynamically loadable code. Nevertheless, other programming languages could be also used.

instantiated as well as the identifier of the new instance to be created. By separating the two concerns of determining instance identifiers and the actual instantiation, in JOpera it is possible to decouple an expensive operation – the creation of an instance, which can be carried out in parallel, notwithstanding – from the less expensive sequencing of instances, which can however be a potential performance and scalability bottleneck.

The result of the instantiation procedure is a newly created (possibly persistent) image of a process instance, which fully describes the state of the execution of a process. This information contains both meta-data about processes and their tasks (e.g., their current execution state), as well as the values of all of the corresponding input, output and system parameters. As an optimization, the initial values of these parameters are immediately set from the user-provided default values. Similarly, in case of parameters bound to constants, such bindings are implicitly evaluated at compilation time, so that at runtime, when the instance is created, no extra work is required.

Some of the information included in a process instance is used by the navigation procedure, i.e., the state of the tasks and the values of the parameters are read upon the evaluation of the activators and start conditions associated with the dependent tasks. Furthermore, in order to guarantee their recoverability, the state of the process instances also contains the data produced and consumed by the services that are invoked as part of the process. In addition to enable the recovery of such instances, storing all parameter values is also very useful for monitoring and debugging purposes, as the user can inspect them interactively.

Executing instantiation There are several options to implement the instantiation procedure. Depending on the architecture and on the reliability guarantees provided by a process support system, instantiation of new processes can be delegated to the database subsystem, where the required tables and tuples to store the state information of the new instance are created [21, 98, 145]. Although this approach can be optimized to reduce the cost of process instantiation by leveraging the functionality offered by a specific database system, there are some limitations concerning the flexibility of the system as the instantiation procedure becomes dependent on the underlying persistent storage technology. On the one hand, when switching to a different database, in a worst case scenario, the entire instantiation procedure may have to be reimplemented. On the other hand, offering support for multiple storage platforms[9] requires a large maintenance effort, as every modification to the process instance model affects all implementations.

To address this portability issue, while keeping the possibility to do optimizations, in JOpera we have followed a strategy based on:

1. the definition of a storage subsystem interface with a higher level of abstraction[10];

[9] including volatile storage, as not all usage scenario require the expensive recoverabilty provided by persistent storage.

[10] See Section 34 on page 155 for more information on this.

2. the generation – at compilation time – of a process-dependent instantiation procedure based on the abstract storage subsystem interface.

Therefore, in JOpera not only the instantiation procedure has been decoupled from the actual storage technology but there is no generic instantiation code that interprets the structure of the process in order to extract the information that defines the state of the new instance at run-time. Instead, the compiler prepares such image beforehand and uses it to generate the process-specific instantiation code which can be run on any implementation of the actual storage subsystem.

6.5.3. Task State Diagram

While it is processed by the system, a task instance goes through several execution stages [130]. These can be modeled by the finite state machines of Figures 6.7 and 6.8. Defining a model of the task execution state is useful for a variety of purposes:

1. The user receives clear feedback about what is happening within the system. Knowing the current state of a task is very useful for monitoring purposes, for detecting failures and for knowing what are the options regarding the interaction with the task. For example, it is not possible to suspend a task that has already been finished.

2. As the state of the execution of a task is stored persistently, this information can be used for recovery purposes in case of internal system failures. More precisely, depending on the last known state of a task, different recovery actions may need to be carried out.

3. The navigation algorithm generated by the compiler for a given process is implemented by connecting together each task's state machine as specified by the control flow graph of the process. The execution of the navigation algorithm is triggered after each state transition in order to execute a set of actions corresponding to the state of the current task, or to fire state transitions in the subsequent tasks.

4. The most important state transition events are logged in order to provide a history of the execution of the process instances. This information can be then mined to gather profiling information and analyzed to detect bottlenecks in the structure of the process.

Visible state diagram of task instances

As a first approximation, we introduce the simplified state diagram of Figure 6.7 which is used to give feedback to the user about the current state of the execution of each task. By default, during instantiation, a task is created in the *initial* state. When the activator associated with the task is triggered, depending on the outcome

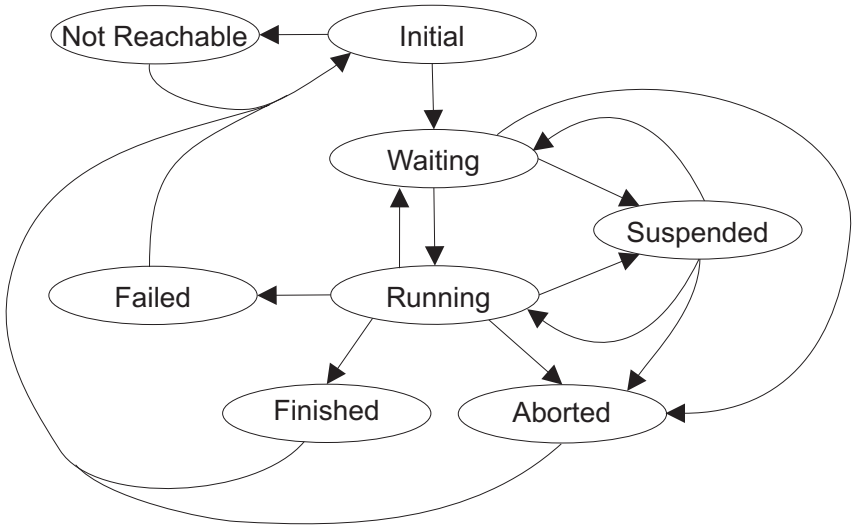


Figure 6.7.: *Simplified state diagram of a task instance as it is seen by the user*

of the condition's evaluation the state of the task may go to *not reachable* – if the condition evaluates to false – or *waiting* – otherwise. The *waiting* state indicates that the task is ready to be executed and the system is looking for a suitable provider in order to invoke the corresponding service. During the invocation of the service, the state of the task is set to *running*. In case the service provider becomes unavailable before the service invocation is completed, it is possible to automatically reschedule the invocation by resetting the state of the task to *waiting* so that an alternative provider can be chosen. Once the invocation has completed, the state is set to *finished* or *failed* depending on whether an error has been detected[11].

The user may interact with a task in different ways depending on its state. If the task is *waiting* or *running* the user may suspend it, i.e., block its execution until it is manually resumed. Furthermore the user may forcefully terminate the execution of a task, by setting its state to *aborted*. Similarly, once a task has reached one of its final states (*not reachable*, *finished*, *failed*, or *aborted*) it is possible to manually restart its execution, by resetting its state back to *initial*. However, when a task reaches one of the final states, the corresponding control flow dependencies will be triggered and the execution of the subsequent tasks may begin[12].

Internal state diagram of task instances

More in detail, the navigation code generated by the compiler uses some additional internal states (Figure 6.8) to address the following issues:

1. Internally, the *suspended* state is paired with the *dequeued* state in order to

[11] See chapter 4 on page 45 for more information on how failures detection is modeled for different types of services.

[12] There is a close relationship between the four final states of a task instance and the four possible control flow dependencies described in Section 3.4 on page 25

	$S \rightarrow S'$	Description
1.	$\rightarrow$*initial*	A task is in the *initial* state when it is first created.
	initial$\rightarrow$*waiting*	Before a task is queued to be executed the values of its input parameters are fetched as specified by its incoming data flow bindings.
	initial$\rightarrow$*not reachable*	If the condition associated with the task evaluates to false, the execution of the task is skipped.
	waiting$\rightarrow$*running*	Once the actual execution of the task begins, the address of the chosen service provider is recorded for recovery purposes.
	running$\rightarrow$*waiting*	If the provider currently executing the task becomes unavailable, the task is automatically rescheduled so that it can be submitted to an alternative provider.
2.	*running*$\rightarrow$*finishing*	A task has completed its execution.
	finishing$\rightarrow$*finished*	The outgoing data flow bindings of a task are evaluated before navigation can proceed.
	running$\rightarrow$*failed*	A task has failed its execution.
3.	*waiting*$\rightarrow$*dequeued*	If the user suspends a *waiting* task, its state is immediately set to *dequeued* as there is no remote provider to contact for actually suspending the task.
	dequeued$\rightarrow$*waiting*	When the user resumes a *dequeued* task, it is immediately put back into the queue.
	running$\rightarrow$*suspending*	The request to suspend a *running* task is submitted to the dispatcher.
	suspending$\rightarrow$*running*	The dispatcher cannot suspend a task.
	suspending$\rightarrow$*suspended*	The dispatcher has suspended the task.
	suspended$\rightarrow$*resuming*	The request to resume a *suspended* task is submitted to the dispatcher.
	resuming$\rightarrow$*suspended*	The dispatcher cannot resume a task.
	resuming$\rightarrow$*running*	The execution of the suspended task has been resumed by the dispatcher.
4.	*waiting*$\rightarrow$*aborted*	The execution of a task which was not yet started has been prevented from happening.
	dequeued$\rightarrow$*aborted*	A *dequeued* service invocation was killed.
	running$\rightarrow$*aborting*	A request to interrupt a *running* task is forwarded to the dispatcher managing its execution.
	aborting$\rightarrow$*running*	The dispatcher cannot kill a task.
	suspended$\rightarrow$*aborting*	A request to stop a *suspended* task is forwarded to the dispatcher.
	aborting$\rightarrow$*aborted*	The dispatcher has successfully interrupted the execution of the task.
5.	*finished*$\rightarrow$*initial*	A successfully completed task is restarted.
	failed$\rightarrow$*initial*	The failed invocation of a task is retried.
	aborted$\rightarrow$*initial*	The interrupted execution of a task is repeated.
	not reachable$\rightarrow$*initial*	The state of a skipped task is reset so that its condition can be re-evaluated.

Table 6.1.: *Task State Transitions*
Actions carried out for each state transition of a task instance

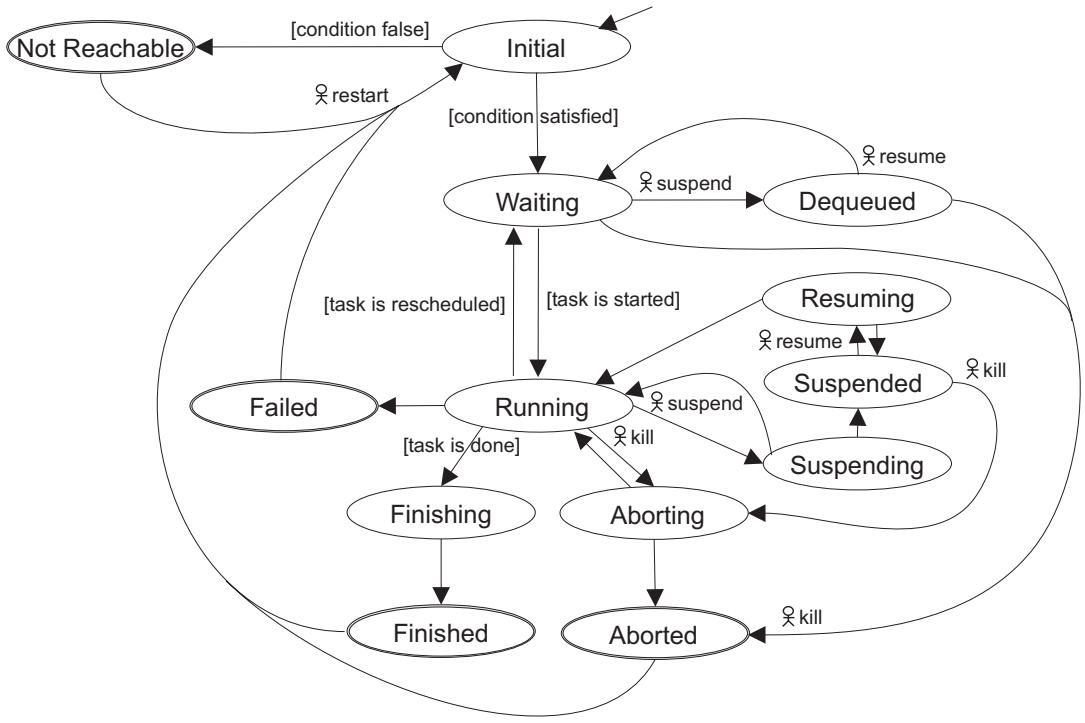


Figure 6.8.: *Full state diagram of a task instance as it is followed by the code generated by the compiler*

know whether upon resuming it, the task should go back to the *running* or to the *waiting* state, respectively.

2. Three of the additional intermediate states (i.e., *aborting*, *suspending*, *resuming*) have been introduced in order to capture the asynchronous interaction between the various JOpera system components. More precisely, the intermediate state is entered upon the receipt of a request by the user and is exited after the requested action has been completely carried out by a potentially remote part of the system.

   For example, if the user suspends a *running* task, upon receipt of such request the state of the task is set to *suspending* and the navigator initiates the corresponding action by contacting the dispatcher, which could be running on a different machine[13]. Depending on the type of the service that is being invoked, the dispatcher attempts to suspend its execution and signals with another event the result of the action. If it was possible to suspend the task, its state finally goes to *suspended*, otherwise is reset back to the original *running* state and the user is notified about the error. In fact, depending on the type of service to be invoked as part of the execution of a task it may or may not be possible to perform such types of interaction[14].

[13]Please turn to Section 7.3 on page 154 for more information on JOpera's architecture

[14]Although it is possible to signal a UNIX process and pause its execution, it is not feasible to

State	Condition
initial	by default a new process instance is created in this state, which is left as soon as the navigation algorithm is executed once on it.
finished	all tasks are either *finished* or *unreachable*[15].
waiting	at least one task is waiting.
suspended	at least one task is suspended.
running	at least one task is running.
aborted	at least one task has been aborted and this event was not handled by another task.
failed	at least one task has failed and its failure was not handled by another task.

Table 6.2.: *Process State Definition Rules*
These rules define how the state of a process instance is aggregated from the state of its tasks

3. The *finishing* intermediate state is reached after a task has successfully completed its execution. During this state, the values of its output parameters are copied as specified by its outgoing data flow bindings. After these data transfer operations are completed, the task is finally set to the *finished* state so that the navigation algorithm can be triggered and eventually the execution of the dependent tasks started.

Before discussing the state diagram of a process instance as a whole, in Table 6.1 we give a precise specification of what kind of actions are to be performed for each one of the state transitions (from state S to state S') shown in Figure 6.8. The first set of transitions is used to begin the execution of a task. The second set of transitions describes the possible outcomes of a service invocation (success or failure). The third group of transitions models how the user can suspend and later resume the execution of a task instance. Similarly, the fourth group models the interruption of the execution of a task instance, while the last transitions deal with restartable tasks.

State of the execution of a process instance

The overall state of the execution of a process instance depends on the state of its component tasks. In order to aggregate the state of all tasks of a process into one value, we specify a set of rules, which define the state of a process instance as a function of the collective state of its tasks. In order to simplify the rules, we assume that they are evaluated with the priority corresponding to the order in which they are given in Table 6.2.

[15] temporarily suspend the remote invocation of a Web service

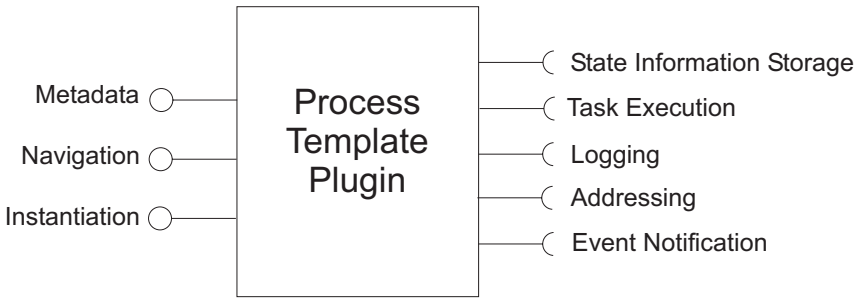


Figure 6.9.: *Process Template Plugin Interface*
UML Component Diagram describing the interface of a process template plugin in terms of the provided (left) and required (right) services

6.5.4. Process Template Plugin Interface

In the previous sections we have presented the assumptions that are made by JOpera's compiler concerning the state of the task and process instances, as well as what the compiler should produce given a process description: the executable code responsible for performing navigation and instantiation. Before showing in detail an example of the produced Java code, we describe the interface between a so-called process template plugin and the rest of the JOpera kernel (Figure 6.9). This way, we clarify the interaction points between the system and the process navigation and instantiation code and motivate the design of part of the JOpera process execution kernel, which is presented in detail in the next chapter.

As a general note, it is a very interesting design problem, to define such an interface in terms of what the plugin provides and what the plugin requires, as these choices can affect both the flexibility and the performance of the resulting system.

First of all, a process template plugin provides the following functionality to the rest of the JOpera system.

1. *Instantiation* – Creation of new process instances for the given template.

2. *Navigation* – Execution of one step of the navigation algorithm over any of the previously created process instances.

3. *Metadata* – Information about the template. This includes the ability to query the plugin to retrieve both the metadata contained in the original OML process (name, description, and author), as well as the compilation date and, in principle, additional versioning information both about the process description and the compiler which produced the code.

In order to implement this functionality, a process template plugin requires and uses a small and well defined set of facilities provided by the rest of the JOpera kernel. They are briefly introduced in the rest of this section and they will be described more in detail in the next chapter.

1. State information storage services are used both by the instantiation code – to create an image of a new process instance – and by the navigation code, to implement data flow bindings and evaluate conditions.

2. Task execution requests are queued by the navigation code, once it determines that a particular service should be invoked.

3. Logging and execution profiling facilities are used by the navigation code to instrument the execution of a process instance in order to measure its performance.

4. Addressing data structures are needed to identify process and task instances, as well as their parameters, so that their values can be accessed within the state information storage services.

5. An event notification mechanism is also a requirement, as the navigation code is called for every state transition event of a certain process (and task) instance. Furthermore, (both synchronous and asynchronous) sub-process calls are also implemented on top of event notification.

Example 6.1: Compiling the Stock Quote Currency Conversion Process

In this example we show the most important sections of the Java executable code produced as a result of the compilation of the process shown in Figure 4.6 on page 57 within Example 4.1 on page 55. Although this is a small example with only three tasks, with it we can already illustrate the most important aspects of process compilation. For completeness, the full listing of such code can be found in Appendix B.

The `StockQuoteConvert` process shown in Figure 4.6 on page 57 is compiled into the `TStockQuoteConvertTemplate` class. This class has six methods, four of which (`getName`, `getAuthor`, `getDescription`, `getCompileDate`) are simply used to store metadata about the plugin, so that it can be correctly identified. Furthermore, the navigation code is contained in the `Evaluate` method, while the instantiation code is in the `SetupImage` method. In the following we will point out the most important code pattern in both of these methods, and explain how they relate to the original structure of the process.

Instantiation code

The `SetupImage` method is called once for every new instance of a process.

```
218   public void SetupImage(TID Context, Map Params)
```

It receives two parameters: the first (`Context`) identifies the new instance, while the second (`Params`) contains the values of user-provided process input parameters.

In order to create a new instance, the method uses two different facilities offered by the JOpera platform:

1. The `SetupParam` method is used to allocate a new parameter, identified by its Address (composed of instance, box – or namespace, which can hold one of the following values: `Box.Input`, `Box.Output`, or `Box.System` – and name). Furthermore, this function also takes the initial value for the parameter.

2. The `SetupSystemBox` is used to allocate a set of system parameters used to store metadata about a certain process (or task) instance. This method allows to keep the compiler independent of the actual set of system parameters that are used, enhancing the flexibility of the system. However, the system parameters whose values depend on the process description are still explicitly allocated by the compiler.

In the example, the following code allocates the system parameters for the process instance:

```
220     SetupSystemBox(PROC(Context));
```

Furthermore, the user defined input and output parameters of the process are allocated. Whereas the output parameters are set to their default values, the values of the input parameters are read from the ones provided by the user, which are stored in the `Params` parameter of the `SetupImage` method.

```
221     SetupParam(PROC(Context), Box.Input, "symbol",Params.get("symbol"));
222     SetupParam(PROC(Context), Box.Input, "country",Params.get("country"));
223     SetupParam(PROC(Context), Box.Output, "quote","");
```

Similarly, for each task, a set of calls to the `SetupParam` and `SetupSystemBox` are issued. In addition to the user defined input and output parameters, also a set of system parameters related to the tasks are prepared.

```
225     TID Context_TASK_getStockQuote = TASK(Context, "getStockQuote");
226     SetupSystemBox(Context_TASK_getStockQuote);
227     SetupParam(Context_TASK_getStockQuote,Box.System,Box.Name,"getStockQuote");
228     SetupParam(Context_TASK_getStockQuote,Box.System,Box.Type,Box.Activity);
229     SetupParam(  Context_TASK_getStockQuote,
230           Box.System,
231           Box.Prog,
232           "StockQuotePort_getStockQuote");
233     SetupParam(Context_TASK_getStockQuote,Box.System,Box.MaxRestart,"0");
234     SetupParam(Context_TASK_getStockQuote, Box.Input, "symbol", "");
235     SetupParam(Context_TASK_getStockQuote, Box.Output, "Result", "");
```

Navigation code

The navigation code is contained in the `Evaluate` method which is intended to be called for every state transition event regarding a certain process instance. The process instance for which such an event should be evaluated by the navigation algorithm is identified by the `Context` parameter of the method.

```
35    public void Evaluate(TID Context) throws MemoryException
```

Before we present in detail how the control flow and data flow description is mapped to the executable code of the `Evaluate` method, it should be noted that the navigation code, which can be considered as the Java implementation of the set of ECA rules, does not make

any assumptions about the order nor the duration of the tasks, although it strictly enforces the control flow structure of the process.

More precisely, the `Evaluate` method is called multiple times to execute each navigational step of a process instance. Therefore, by setting different values for the `Context` parameter, it is possible to call multiple times the same method to interleave navigation on multiple instances of the same process. Furthermore, the method returns as soon as one step of the navigation algorithm has been performed, although the execution of the process instance may have not yet completed.

By inserting conditional code that checks the current state of the instance, it is possible to enable the asynchronous interaction between the `Evaluate` method and the task execution, which is triggered as a result of navigation from within the `Evaluate` method but carried out asynchronously in different parts of the system. This approach is needed to support parallelism in the control flow of a process, as multiple task execution requests may be issued concurrently and, similarly, it is necessary to deal with tasks of variable duration, for which it is not possible to foresee the order of completion.

Due to the asynchronous nature of this interaction, the current state of the instance identified by the `Context` parameter must be reconstructed during each call to the navigation code. To do so, a set of local variables of type `State` are used to read the state of the process (and later of its tasks) from the state information storage facilities, identified in the code by the interface called `Memory`.

```
39  State State_PROC;

46  State_PROC = Memory.getState(Context_PROC);

100 State State_getStockQuote = Memory.getState(Context_TASK_getStockQuote);
101 State State_getExchangeRate = Memory.getState(Context_TASK_getExchangeRate);
102 State State_Multiply = Memory.getState(Context_TASK_Multiply);
```

After reading the state of a process instance it can be decided what needs to be done, if navigation has been invoked on a newly created process instance (in state *initial*), the tasks without any control flow dependency should be started.

```
48      if (State_PROC == State.INITIAL)
```

The example process has two of such tasks, which should be executed in parallel. Before a task execution request can be issued, however, it is necessary to prepare its input data according to the data flow bindings of the process description. In this case, the `getStockQuote` task has one input parameter called `symbol` which should contain the value of the process input parameter with the same name. This parameter copy operation is described by the corresponding data flow binding and is executed through the following Java code:

```
52        Memory.Copy(MakeAddress(Context_PROC, Box.Input, "symbol"),
53            MakeAddress(Context_TASK_getStockQuote,Box.Input,"symbol"));
```

If a task had multiple incoming bindings, for each of the bindings there would be a similar `Memory.Copy` statement, that instructs the state information storage subsystem to copy a value from a source to a destination address. Addresses are used to identify a parameter of a certain process (or task) instance.

After all incoming bindings have been executed, the input data for the task should be fetched and packaged as part of the task execution request.

```
56        InputParams.put("symbol", Memory.Load(MakeAddress(
Context_TASK_getStockQuote,
57                                      Box.Input,
58                                      "symbol")));
```

Finally, the task can be started.

```
61        Exec.Start(Context_TASK_getStockQuote, InputParams);
```

It is important to point out that, in general, the call to `Exec.Start` returns as soon as the task execution request has been submitted to the task execution scheduler. Therefore, it is possible to execute multiple tasks of the same process instance in parallel, as they are started asynchronously.

In fact, this is what the navigation code of the example does, as it continues with similar code used to evaluate the incoming data flow bindings of the `getExchangeRate` task and to fetch its input data for the corresponding execution request.

```
78        String p_country1 = (String) InputParams.get("country1");
79        String p_country2 = (String) InputParams.get("country2");
80
81        if (!(p_country1.equals(p_country2)))
82        {
83          TimeStamp(Context_TASK_getExchangeRate, Box.StartTime);
84          Exec.Start(Context_TASK_getExchangeRate, InputParams);
85        }
86        else
87        {
88          Memory.setState(Context_TASK_getExchangeRate,State.UNREACHABLE);
89        }
```

However, in the original process description this task is associated with a condition that controls for which parameter values the task may be actually started. The code to implement such start conditions first fetches the parameter values that should be compared into local variables and then uses them to build a Java expression corresponding to the original condition. If this expression evaluates to true, the normal task execution request code is executed. Otherwise, the state of the task which is not executed is set to *not reachable*.

After all tasks without incoming control flow dependencies have been started (or skipped) the state of the process is set to *running*, and the first execution of the `Evaluate` method terminates.

```
90        Memory.setState(Context_PROC, State.RUNNING);
```

As soon as one of the two task running in parallel finishes, the `Evaluate` method is called again and, since the state of the process instance is no longer *initial*, it will not repeat the previously described code, but instead continue with the rest of the navigation, which involves the third (and last) task of the process.

More precisely, after the execution of a task has completed, its state is set to *finishing*, if its execution was successful. In this case, before the navigation can continue to look for new tasks to start, the results of the service invocation must be stored persistently in the state information storage subsystem.

```
106           Memory.Store(MakeAddress(  Context_TASK_getStockQuote,
107                        Box.Output,
108                        "Result"), (String) Results.get("Result"));
```

Furthermore, in case there are outgoing bindings from a task to the process output parameters, these must also be carried out, in a way very similar to the incoming bindings.

```
109           Memory.Copy(MakeAddress(Context_TASK_getStockQuote,
110                       Box.Output,
111                       "Result"),
112                 MakeAddress(Context_PROC, Box.Output, "quote"));
```

Only at this point, the state of a task is set to *finished* to allow the navigation algorithm to continue. As an optimization, to avoid exiting and reentering the `Evaluate` method, the cached state of the task is also immediately changed.

```
114           Memory.setState(Context_TASK_getStockQuote, State.FINISHED);
115           State_getStockQuote = State.FINISHED;
```

As specified by the control flow graph of the process, only if both the `getStockQuote` and `getCurrency` tasks have completed their execution the third task can be started.

```
130           if ((State_getStockQuote == State.FINISHED)
131           && (State_getExchangeRate == State.FINISHED))
```

Once both control flow dependencies are satisfied, the incoming data flow bindings of task `Multiply` can be evaluated, its input parameters are fetched as previously described.

After its input parameters are ready, the task can be started.

```
157             Exec.Start(Context_TASK_Multiply, InputParams);
```

On the other hand, if it is not possible to execute the task, because one of its predecessors in the control flow graph will not satisfy its activator, the state of the task is set to *not reachable*.

```
171             Memory.setState(Context_TASK_Multiply,
172                     State.UNREACHABLE);
```

The last part of the `Evaluate` method contains the code responsible for determining the state of the process instance as an aggregation of the state of its component tasks. After one navigation step has been concluded and all of the state transitions of the tasks have been determined, it is possible to apply the rules specified in Table 6.2 to compute the overall state of the process instance. In this example we focus on the process termination rules, that control whether a process has *finished* or *failed*.

The following statements control when the process is considered to be *finished*. It should be noted that the expression on line 191 refers only to the state of the tasks without outgoing control flow dependencies.

```
191         if (((State_Multiply == State.FINISHED) || (State_Multiply ==
State.UNREACHABLE)))
192         {
194               Memory.setState(Context_PROC, State.FINISHED);
195         }
```

These statements control when the process fails. As described earlier, this happens only if at least one task has failed and its failure has not been handled by another task.

```
197         if ((State_getStockQuote == State.FAILED)
198         || (State_getExchangeRate == State.FAILED)
199         || (State_Multiply == State.FAILED))
200         {
201           if (State_PROC != State.FAILED)
202               Memory.setState(Context_PROC, State.FAILED);
203         }
```

Finally, the following code is executed the last time the `Evaluate` method is called for a given process instance. More precisely, this code is used to gather the results of a process, and, in case the process was started as a sub-process call from within another process, the `Completed` method notifies the caller about the outcome of the execution of the invoked process.

```
206       if ((State_PROC == State.FINISHED) || (State_PROC == State.FAILED))
207       {
208         Results.clear();
209         Results.put("quote", Memory.Load(MakeAddress(  Context_PROC,
210                                  Box.Output,
211                                  "quote")));
212
213         Completed(Context_PROC);
214       }
```

6.6. Discussion

This chapter can be interpreted in terms of the model driven architecture (MDA) paradigm. Model driven architecture is an approach to software construction that, in the extreme, strives to replace the manual production of program code by the automatic generation of such code starting from formal specifications, which model the design of a system [127]. Just like compilers helped to replace assembly level programming with programming languages at a higher level of abstraction, compilers should be also used as a tool to transform the model of a system into the corresponding program code.

Very similar ideas can be applied to the domain of process modeling. In Chapter 5 we have defined a language (or meta-model) for modeling processes. In this chapter we have presented what are the options as far as the transformation of such processes in other, executable representations is concerned. More precisely, given a process model written in an OML document, model checking techniques can be applied to ensure its consistency. Once such model is deemed to be correct, it is possible to define and apply, up to a certain degree, reversible mappings between different process meta-models (e.g., exporting a BPEL process from one defined in OML). This kind of transformation can be used to prepare a process to be interpreted by a given process execution engine. Also regarding process execution, one-way compilation is a very useful technique, in order to transform a process model into

executable code, as we have shown in the last part of this chapter.

Looking outside of the process modeling and execution area, compilation and interpretation are two standard alternatives, as far as the execution of traditional programming languages is concerned. Each one has its strenghts and weaknesses, and recently the distinction has been somewhat blurred by just in time compilation techniques, which attempt to synthesize the best of both worlds [50].

One of the advantages of interpretation is that source code can be directly executed, without an intermediate, time-consuming, static compilation step. Furthermore, as there is no distinction between compile-time and run-time, all of a series of dynamic techniques can be applied to handle variations and adaptations (including late binding) in the execution environment, which would be more difficult in a pure-compilation approach, where most decisions are taken statically.

Along this line, one of the typical reasons for using interpretation to execute process descriptions lies in the possibility of dynamically changing the definitions of the processes on the fly, in order to handle exceptional cases which were unforeseen at the time of the process was defined. Thanks to interpretation, where, at run-time, process descriptions are not considered a read-only data structure, dynamic workflows environments exhibit a *liveness level three* [226], where users may modify the structure of a process as it is running while the underlying interpreter ensures the consistency of the changes.

In the context of this dissertation, processes define scripted interactions between services, which involve – in most cases – completely automatic execution. That is, in JOpera production quality processes run without neither human supervision nor interaction, which are instead the rule in the case of business process automation. Therefore, it becomes less clear how an approach based on interpreted, dynamic workflows could be useful, in order to allow developers to change their active processes manually. Conversely, in our opinion, the evolution of processes, a kind of software artifact, should happen through an orderly change review process based on version control systems, like in most modern software engineering approaches. Therefore, by extending process definitions with versioning information, in JOpera the compilation of processes does not interfere with the possibility of customizing and redefining a process to suit particular purposes which were not foreseen at the time it was first defined. Before it can be executed, a modified process, like any other process, must be compiled and – in addition to its name – a version identifier can be used to distinguish the particular revision when starting it. This way, it is possible to concurrently run different versions of a process, as version numbers are also transparently used to identify the process plugin generated by the compiler.

One of the difficulties of applying compilation to processes lies in the fact that processes are intended to be executed multiple times concurrently. Also with compilation applied to traditional programming languages, where the source code of a program is transformed into object code, and then linked to build an executable program, it is possible to execute multiple instances of the same program concurrently, as long as the limits within the operating system are not exceeded. The JOpera system, like other process support systems, is intended to scale beyond such limits

and to be able to run a very large number of concurrent processes. Therefore, compilation of process descriptions to executable code, intended to be run directly on top of the operating system is not an option, although the design of the compiler and its output would probably have been simpler because the generated code would have been used to run only one process instance delegating all multi-processing issues to the operating system's scheduler.

As a consequence, in the design of JOpera's compiler, the Java executable code generated for a given process description is not a stand-alone program, to be run directly in a Java virtual machine. Instead, as we have discussed in Section 6.5, processes are compiled into plugins that are dynamically loaded into JOpera's kernel to manage the creation and the concurrent execution of multiple process instances. To do so, the executable code for a process generated by the compiler is completely stateless, as it only contains the process dependent navigation and instantiation logic – corresponding to the structure of the process. To determine the next services to be invoked, the code reads the current state of the execution of a given instance as input. Nevertheless, by configuring JOpera's process execution kernel appropriately and linking it with the process code, it is still possible to execute a single instance of a process as a stand-alone program, as we will show in the next chapter.

7. Architecture

In this chapter we present the architecture of the JOpera system, which provides an integrated set of tools to support the development and the execution of the processes written in the JVCL language presented in Chapter 3. The tools composing JOpera can be organized following the life cycle of a process in three main categories: design-time, compile-time and run-time. While in the previous chapter we have already discussed the features and the architecture of JOpera's compiler, in this chapter we present the architecture and usability features of the visual development environment (design-time) and also the architecture and deployment options regarding the process execution kernel (run-time).

7.1. Motivation

The main purpose of the JOpera system is to provide the user with an integrated and flexible environment for creating, editing, checking, compiling, deploying, executing, monitoring and profiling the processes written in the JVCL language. The main goals for creating the JOpera system were:

- *Ease of use.* First of all, we believe that a visual language – as opposed to an XML based representation – can greatly improve the understanding of a process built by drawing connections between different component services [187]. However, employing a visual language is not enough to claim the usability of the system supporting it [191]. In this context, *visual scalability* is an important issue, as the understandability of a large diagram may decrease as its size increases [35]. Therefore, we will discuss how the visual notation and the supporting environment address this important usability factor.

- *Flexibility.* Flexibility is a key aspect of the JOpera system for two reasons. First of all, flexibility is very important in supporting and dealing with diffent types of components. Not only an open (and flexible) component metamodel is required, as discussed in Chapter 4, but the underlying architecture of the system should also provide the necessary extension points to plug in the adapters which efficiently map JOpera's high level model to the actual protocols used to perform a service invocation.

  Furthermore, the JOpera process execution kernel provides a flexible execution platform for the processes written in JOpera Visual Composition Language

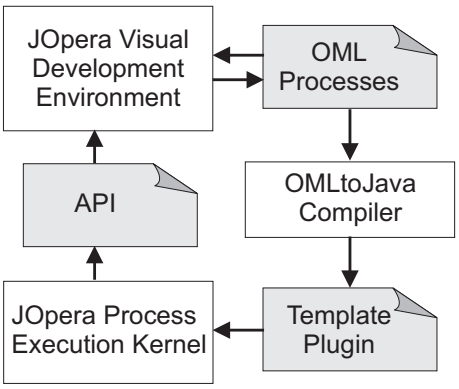


Figure 7.1.: *Overview of the JOpera Visual Development Environment and the JOpera Kernel*

which can be tailored to different quality of service guarantees, both in terms of performance, reliability and scalability, as well as portability to different environments, in which the system can be deployed. More precisely, as we will present in the second part of this chapter, JOpera's kernel can be deployed in several different configurations. For example, in a light weight configuration, it may run embedded as a library in other Java applications. At the other end of this spectrum, JOpera may run independently, as a cluster-based process execution engine which can scale to handle large workloads.

Figure 7.1 depicts the relationship between the JOpera Visual Development Environment, the Compiler and the Process Execution Kernel. The processes defined in the JVCL language are created and edited using the development environment. As we have discussed in the previous chapter, once the processes are complete and ready to be executed, they are first compiled into Java and the resulting process template plugins are then dynamically loaded into the kernel for execution. Once a new process instance has been started, its execution is managed by the kernel, which may be run independently of the development environment. However, multiple users may connect a development and monitoring environment to an existing kernel to monitor the activity and the progress of their processes.

In the first part of this chapter we describe JOpera's visual development and monitoring environment, while in the second part we focus on the design of the flexible architecture of the process execution kernel.

7.2. Visual Development Environment

In this section we introduce JOpera's visual development environment, an integrated set of tools we have built to support the JOpera Visual Composition Language. First we give an overview about the development cycle from the user's point of view. Then we concentrate on its visual scalability features and conclude by describing its architecture.

7.2.1. Development cycle

The whole lifecycle of a process can be managed with the JOpera visual process development environment. First of all, Web services and other component types can be imported into the service library as reusable components. The user can browse through it, select a set of services and drag and drop them into the data flow graph of a process. At this point, the development environment can automatically suggest data flow connections between parameters of matching names and data types, or, for example, assist the user in building the appropriate data conversion filters between mismatching parameters. The control flow graph of the process is also automatically kept consistent with the data flow graph, so that the user can look at it to get an overview over the order of invocation of the services or to add additional constraints. When deleting a control flow dependency all of the corresponding data flow bindings are removed. Conversely, whenever a new data flow binding is established, the corresponding control flow dependency is added. The user is notified with optional warning messages of the consequences of these actions, which otherwise are carried out in a transparent manner[1].

Once all services have been connected the process is compiled to Java executable code and uploaded to a JOpera runtime environment for execution. During compilation various consistency checks of the process model are carried out and the user is notified with a list of errors (e.g., parameters of incompatible types are connected) and warnings about potential problems (e.g, an input parameter has been left disconnected).

After a successful compilation, the user may start multiple concurrent instances of a process, which are managed by the same runtime environment.

The user may keep track of the progress of a running process (an *instance*) through various monitoring tools, which provide both the ability to inquire about the state of a process instance, interact with it and receive notifications whenever a state change occurs. The progress of a running process can also be visually monitored by watching the color of task boxes, indicating their execution state, and by clicking on data parameters to inspect (and modify) their content. The user may interact with a process and its tasks to abort, pause, continue and restart their execution at will.

Once a process terminates its execution, the system keeps its state in a history database, which includes both the content of all parameters as well as profiling information with measurements about the execution time of each task. This information can later be analyzed, e.g., for performance optimization.

Moreover, in order to optimize response time and resource utilization, JOpera checks incoming process instantiation requests against the history database. To service similar requests, it may reuse results already computed. The same kind of optimization can be performed with respect to currently running processes, avoiding to perform duplicate work. Caching results in the history database can lead to overflow, therefore JOpera provides various garbage collection policies. The users

[1] See Section 3.4 on page 25 for more information on how the relationship between the data flow and control flow graphs of a process has been defined in the language.

may set an expiry date on their process instances and, similarly to [139], system administrators may delete process instances no longer needed. The system itself may also purge from the database results that are least frequently reused if more space to store newer results is needed.

Finally, once its development and testing has been completed, production-level processes can be published as Web services [198].

7.2.2. Visual scalability

One of the advantages of using a visual programming language is that the data and control flow of a process can be specified directly by drawing graphs. In practice, however, some manual effort is required in order to obtain a readable diagram, even for small sized graphs. Thanks to the automatic layout facilities built into the development environment, the amount of work necessary to re-arrange the graph layout is significantly reduced. We have adapted several hierarchical layout algorithms [152] to take into account the syntactical relationships between the graph elements [37]. Furthermore, these algorithms are intended to be used incrementally in order to preserve the user's mental map of the process [163].

Although the automatic layout features already improve the user's productivity, better support is required to visualize realistic graphs having a large number of elements. Therefore our development environment provides the user with other features that increase the scalability of the visual language [35].

1. First of all, thanks to the sub-process construct, parts of the graph may be collapsed into single nodes and the user may easily navigate back and forth between the various levels of nesting. This allows the user to design processes following both a top-down progressive refinement and a bottom-up aggregation approach.

2. Second, the environment provides the ability to create and work with multiple views over the same data flow graph. In this case, the user may easily extract a subset of the data flow graph, for example, to analyze the data flowing through a particular task, or to focus on the tasks receiving data from a certain parameter. This way, the user may navigate interactively through a complex data flow graph and is always presented with an uncluttered view over the relevant information. The development environment also allows the user to edit the data flow graph from any of the views by enforcing the required consistency constraints. For example, when deleting a redundant data flow connection which is present in more than one view, the user will be warned about it and may decide to remove the connection from all views.

7.2.3. Architecture

The JOpera visual development environment has a layered architecture, which provides a common user interface to the rest of the JOpera system. In it, the user has

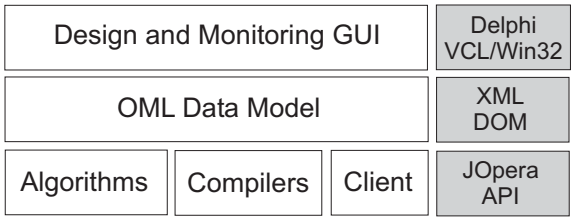


Figure 7.2.: *Architecture of the JOpera Visual Development Environment*

access to a set of tools (editors, model checking algorithms, compilers, and debuggers) which are used to manage the lifecycle of a process. At design time, the visual process editor is used to draw the structure of a process in terms of the control and data flow graph defined as part of the JVCL in Chapter 3. At runtime, the monitoring tool also shows the processes using the same visual notation. Both of these graph visualization and editing tools are based on the Model View Controller design pattern [81]. As shown in Figure 7.2, the design and monitoring graphical user interface is layered on top of the data model, which internalizes the information stored in an OML document and the information extracted from the Process Execution Kernel.

OML Data Model The data model underneath the visual environment is built on top of a Document Object Model (DOM [259]), which provides an in-memory representation of an OML document, a particular kind of XML document. In addition, the DOM API provides an event based notification mechanism for detecting and propagating editing changes (attribute value modifications, insertion an removal of elements) from the model back to the subscribed views. When changes occur, by serializing the DOM into its string representation is also possible to build an undo/redo stack, which buffers the modifications to the model in a generic way.

The interface towards the rest of the system of the OML data model presents a higher level of abstraction, as opposed to the XML DOM API, where a document is represented in terms of its element nodes and their attributes. The development of the editors, the views and the other components can be simplified if these components may use an object oriented data model, based on the inheritance, aggregation and reference concepts defined in Section 5.1 on page 87. This abstraction layer does not need to be manually programmed, as its code can be automatically generated from the description of the data model itself by using advanced model driven architecture [127] techniques.

Design GUI The main purpose of JOpera's GUI is to allow the user to quickly compose services without having to struggle with the XML syntax, which is great for processing automatically semi-structured and self-describing data [68], but not quite optimal, as far as using it for programming is concerned. Therefore, in JOpera's environment the XML syntax of an OML document remains well hidden at all times underneath the design (and monitoring) GUI components. The designer thus

provides the user with a visual environment to enter the information about the processes and draw their structure according to the JOpera Visual Composition Language. When a process model is saved, the corresponding XML serialization in an OML document is produced automatically. Furthermore, model checking features can be easily accessed, in order to provide immediate feedback about the consistency of the process model. In particular, a set of background checks of the OML data model are carried out while the user is working (e.g. ensuring that a new data flow connection can be drawn) while more expensive ones (e.g. generating warnings about disconnected parameters) are triggered manually by the user.

JOpera's graphical user interface is based on the Delphi Visual Component Library running on Windows. These libraries and tools have been selected to build the current version of JOpera's visual environment because they provide a rapid application development environment for quickly building user interfaces, with interesting features, such as visual inheritance and frame-based reuse of GUI widgets and related code, without which the feasibility of the project within the given timeframe would have been questionable. However, in order to overcome current limitations regarding the platforms on which JOpera's development environment can be deployed, a port to the Eclipse [80] environment is well under way at this time.

Algorithms In addition to the basic data management functionality, the data model layer is augmented with a component dedicated to performing consistency checks, transformations and general utility functions on the data stored in an OML model. These algorithms are used as building blocks to support the compilation of the model into other representations (Chapter 6). They are also triggered by changes in the data model in order to ensure its consistency. Here is an non-exaustive classification of some of the most interesting features of this component:

- Consistency checks (Synchronization between control and data flow graphs)
- Referential integrity
- Data extraction and filtering
- Automatic graph layout
- Data flow analysis

Compilers This component is used to transform the OML model into other representations. As we have shown in Chapter 6 in order to execute the processes modeled with OML, they are first compiled to other, executable representations. In addition to the compiler which produces OCR (Section 6.3), BPEL (Section 6.4), and Java (Section 6.5), this component groups also other types of compilers, which do the following:

1. The component responsible for the transformation of OML into its graphical JVCL representation can also considered a compiler. With it, the input of the

transformation is stored within the OML data model and the output is displayed on screen using a graphical notation, which can be edited interactively by the user.

2. A tool is also available, which takes the OML data model and starting from its machine-oriented XML syntax, transforms it into different (HTML, LaTeX) representations intended to be used for automatically producing on-line and off-line user documentation of the content of OML documents.

Monitoring GUI In addition to design-time functionality, the JOpera graphical user interface offers the developer with a set of run-time tools, used for monitoring and debugging the processes. These include an interface for navigating the information contained in JOpera's history database. This way, the user may browse through several representations (e.g. graph-based, table-based, tree-based and key-value list based) of the active processes and tasks and inspect the content of their parameters. With these tools, users may also interact with a running process by aborting, suspending and resuming their execution by sending the corresponding signaling commands to the Process Control API. While a process is suspended, as in most debugging environments, it is also possible to modify the values of the parameters of a process and its tasks.

All of these features are implemented with a similar pattern. Each action performed by the user triggers the invocation of the API, either to initiate a certain operation (e.g. when starting or stopping a process) or to query the system for some information, which is then internalized in the data model before the views can be updated. By subscribing to the event notification API, monitoring clients can automatically refresh the displayed information about the state of a process. This way, it is not necessary for the user to actively (and in some cases repeatedly) click on the "Refresh" button in order to get the latest view on the progress of the execution of a process.

Finally, it should be noted that the monitoring information returned by the invocation of the kernel's API is cached by the client, within the data model. This opens up the interesting opportunity of supporting an off-line mode of operation, where the monitoring tools are used to interactively browse and perform automated analysis over historical information, which does not have to be accessed through the system's API, but is stored locally in the client's cache.

Client In order to provide monitoring and debugging functionality, JOpera's GUI uses this component for interacting with the rest of the system, i.e. with the API of the process execution kernel. In the course of the project, clients for this API have been developed in several programming languages (C++, Java, and Delphi) in order to guarantee the accessibility of the API and make it feasible to integrate external tools with the system.

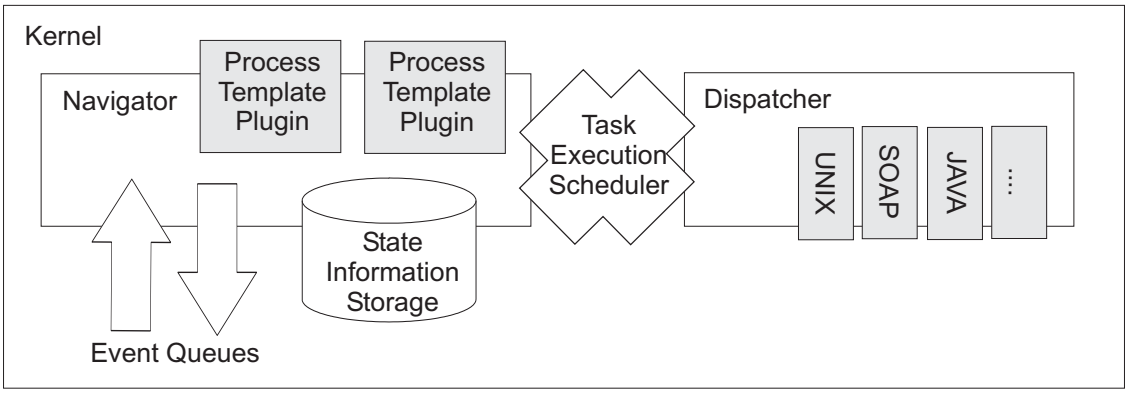


Figure 7.3.: *Architecture of a Monolithic Process Execution Kernel*

7.3. Process Execution Kernel

After presenting the features and the architecture of the user interface of JOpera, in this section we show the core infrastructure necessary to run the processes written in JVCL. As in all client/server architectures, the JOpera's kernel is intended to be connected to the rest of the system (i.e., the user interface) through a well-defined API[2]. This way, one kernel may support the execution of processes submitted by multiple clients of different kinds (e.g., GUIs, Web Browsers). Likewise, as long as the API doesn't change, the actual configuration of the kernel behind it may be freely changed without affecting the presentation layer.

In the following, we first give a detailed presentation of each of the main kernel components, introducing what are the basic options concerning their design and implementation. Then, we define how they interact and we show that, thanks to this approach, the architecture of JOpera's kernel is flexible enough to be deployed in a variety of settings.

7.3.1. Architecture

As depicted in Figure 7.3, the process execution kernel of the JOpera system includes mechanisms to 1) run the navigation algorithm, 2) schedule and 3) dispatch tasks for execution in the correct environment, 4) access and modify state information about tasks and processes, and 5) exchange event notifications triggering the execution of the navigation algorithm itself. After the description put forth in the previous chapter (Section 6.5.4), it should be clear that the navigation algorithm used by the compiler is independent of the actual implementation of these basic facilities.

Navigator

The navigator is the component of the kernel responsible for managing the execution of process instances. To do so, it handles incoming process events, which

[2]See Section 7.5 on page 172 for an overview of JOpera's API

are generally triggered by changes in the state of tasks or represent user requests. When such events occur, for example when the dispatcher has finished executing a task belonging to a certain process instance, the navigator runs the algorithm for deciding what task should be executed next. The navigator also acts as a container for the process plugins generated by the compiler, which embody a process specific version of the navigation algorithm. Upon receipt of events concerning a particular process, if necessary, the navigator dynamically loads the appropriate plugin.

Task Execution Scheduler

This component couples the navigator, generating task execution requests, with the dispatcher, which manages the actual task execution. In a distributed kernel (Figure 7.6), the scheduler receives task execution requests from a number of navigators and forwards them to one out of a set of dispatchers. This is a key component concerning the scalability of the system, as its throughput limits the rate at which tasks can be executed.

Dispatcher

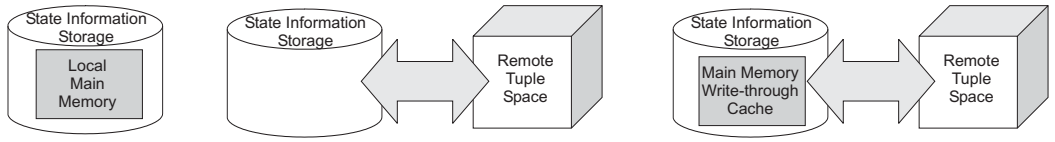
If the navigator is in charge of deciding what tasks should be started next, the dispatcher is the component which actually starts executing the tasks by dispatching them to the appropriate execution subsystem. In order to increase the navigator's throughput[3], the actual task startup operation has been decoupled from the navigation step which triggers it. This way, the navigator may asynchronously issue multiple task startup requests to the task execution scheduler, which queues and forwards them to one or more dispatcher components. Once the dispatcher receives a job it checks what the job's characteristics are and sends it to a matching task execution subsystem[4], which provide an adapter to support the invocation of the services of one of the component types presented in Chapter 4. Once the job's execution has completed, the dispatcher sends an event encapsulating its results back to the navigator.

State Information Storage

This is the component responsible for storing the state information about the process instances. Its design has been influenced by many requirements, such as performance, reliability, and portability across different data repositories. The component's interface supports only a simple, key-value based data model, where the key has been structured as the following tuple `(Process, Task, Instance, Box, Parameter)` and is used to uniquely identify a certain data `Value` across the system. The definition of the key reflects the structure of the information to be stored: a process is composed of a set of tasks, of which there can be many instances. Each

[3] See the following Section 7.3.2 on page 157 for a discussion of the reasons behind this important design decision.

[4] For more information on the internal design of the Dispatcher component, refer to Section 7.4 on page 163.



(a) Volatile Storage (b) Remote Persistent Storage (c) Cached Remote Pers. Storage

Figure 7.4.: *State Information Storage Implementations for the monolithic kernel*

process/task instance has multiple parameters which are grouped into three boxes (or logical namespaces): system, input, output.

The main advantages of this approach are summarized in the following arguments.

- First of all, since the information in the key is neutral with respect to the physical location of the data, it becomes possible to transparently move the data to exploit locality and even replicate it among different physical locations to improve its availability. Furthermore, the hierarchical nature of the key, suggests a natural data partitioning strategy. For example, instances of different process templates can be assigned to different physical data repositories.

- Another advantage is that changes and extensions to the data model of the processes' state information do not affect the storage component, since this low level data representation is mostly independent from the data and metadata that needs to be stored [1].

- Finally, as shown in Figure 7.4 the data layer can be implemented with a wide variety of mechanisms. These range from centralized memory based data structures (such as a hash map), to traditional forms of persistent storage (such as network file systems, or relational databases), distributed storage systems (such as Linda-like tuple spaces [40] like TSpaces [138] or JavaSpaces [76]), or even newer peer to peer storage systems (such as OceanStore [131], Chord [217] and the Cooperative File System (CFS) [59]).

Event Queues

The various kernel components communicate by exchanging event notifications managed by the event queues. Sources for the events consumed by the navigator components are the user interface, other navigators and the dispatchers. Events are sent by the user interface in order to start, stop, and, in general, interact with a process instance. The dispatcher notifies the navigator with an event every time a task has finished its execution. Navigators also exchange events, for example, when a sub-process has completed its execution and navigation over the calling process, managed in general by a different navigator, needs to be triggered. The priority of these three classes of events can be adjusted.

In a distributed kernel, event communication is also important concerning the system's scalability. We compared different implementations of the event queues, each having different scalability properties.

1. First, as a reference, we used a single tuple space server to which all kernel components connect and exchange events by writing and taking tuples. As expected, this centralized event queue quickly becomes a bottleneck if all events sent by multiple dispatchers to a set of replicated navigators need to go through it.

2. Therefore, in a second design, we chose to use a multi-layered approach, by distributing the event queue across all navigator components, with the following heuristic in mind: the navigator responsible for handling the incoming events should be kept as close as possible to the events themselves. This way, the dispatchers may directly send the "task-finished" events to the appropriate navigator. Events which are not sent to a specific navigator still go through the central queue, which, in this configuration, needs to handle relatively less traffic. For example, user generated process startup requests are queued centrally and the corresponding "start-process" events may be retrieved by idle navigators.

To further reduce the communication overhead, as a general rule, events generated by a navigator which can be processed by the same navigator are kept locally and do not need to be sent over the network.

7.3.2. Threading issues

In order to clarify the relationship between the process template plugins generated by the compiler and the overall flow of control within the JOpera kernel, in this section we present more in detail the interaction between the kernel components and its various threads of control.

In the simplest approach, it would seem intuitive to assign one thread of control to the execution of each single process instance. This way, the lifecycle of a process instance corresponds to the lifecycle of a thread in the JOpera kernel: beginning with the instantiation of a process, the thread executes the navigation algorithm until the process instance has completed its execution.

For efficiency and scalability reasons, in JOpera, as opposed to having one thread responsible for managing the execution of one process instance, we have completely decoupled the layout of the physical threads inside the kernel from the execution of the process instances for the following reasons:

- The simple one-thread one-process approach suffers from scalability limitations, as the number of threads (a limited resource) grows linearly with the number of active process instances.

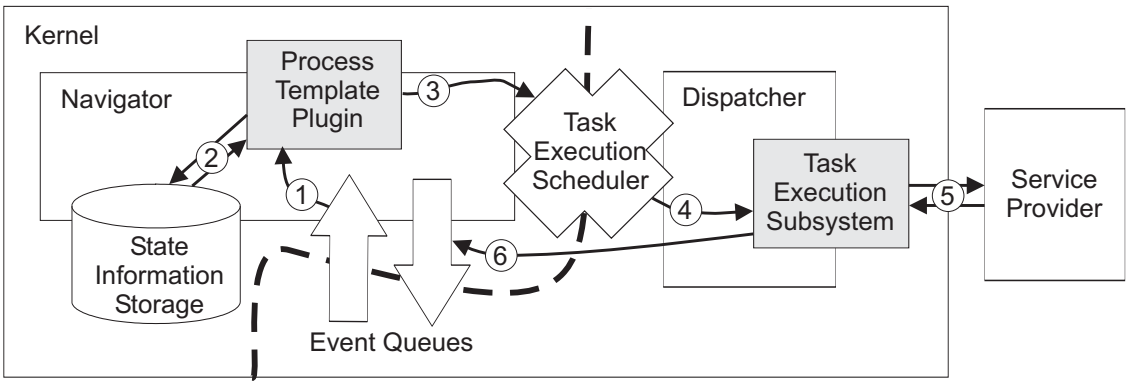


Figure 7.5.: *Main Process Execution Loop*
This figure shows how the kernel components interact during the execution of one step of the navigation algorithm.

- Furthermore, during the execution of a process involving tasks of a long duration, nothing happens most of the time. Therefore, the threads responsible for the navigation of a process would be idle, mostly waiting on notifications that service invocations have completed.

- It is not clear, how, in the simple approach, a single thread would be able to handle the concurrent execution of multiple tasks at the same time.

From these considerations, it becomes evident that, just as physical processors can be shared through the operating system among multiple logical threads, also in JOpera, a very large number of concurrent process instances can be executed by a relatively small number of threads, which[5] can even be dynamically increased or decreased depending on the current workload of the system.

Process Execution Loop

In order to describe how the workload related to the execution of processes can be partitioned and shared among these threads, we consider JOpera's main process execution loop, which defines how the main system components interact (Figure 7.5).

1. An event concerning a certain process instance is retrieved by the navigator, which forwards it to the appropriate process template plugin. Although such an event may also trigger the instantiation of a new process instance, in most cases the navigation algorithm will be executed, as we will assume in the rest of this description.

2. During navigation, state information about the process instance is read and written from the storage component.

[5]as opposed to hardware processors.

3. Furthermore, if the navigation algorithm determines that some services are ready to be invoked, the corresponding task execution requests are issued through the task execution scheduler component.

4. The dispatcher is then contacted to handle the actual service invocation by selecting the appropriate execution subsystem plugin.

5. The subsystem adapts JOpera's invocation request to the actual protocol understood by the service and manages the interaction with the service, according to different interaction patterns[6]

6. Once the service invocation terminates, an event with the results of the invocation is sent to the navigator so that the loop is closed.

As hinted by the previous description and by Figure 7.5, in the design of JOpera's architecture the main process execution loop has been partitioned in two types of actions carried out by different threads. On the left side, the navigator thread is responsible for forwarding events to the appropriate process template plugin and, through the execution of their navigation code, issuing tasks execution requests, which are queued. On the right side, the dispatcher thread is responsible for carrying out task execution requests, by forwarding them to the appropriate task execution subsystem[7], and notifying the navigator of completed service invocations.

Consequently, the queues maintained by the task execution scheduler and the event notification mechanism bind the two main JOpera threads: the navigator, consumer of events and producer of task execution requests; and the dispatcher, consumer of task execution requests and producer of events.

As opposed to the simple threading model, this design provides the following benefits:

- In the smallest configuration, only two threads are necessary to execute any concurrent number of process instances, which may issue any number of concurrent task execution requests.

- In order to increase the task execution capacity of the system, it is possible to replicate the dispatcher thread across multiple processors or even physical hosts by providing the appropriate distributed implementation of the task execution scheduler and the event notification queues.

- Similarly, as long as the appropriate synchronization mechanisms are in place in order to prevent more than one navigator thread to perform navigation over the same process instance, it becomes possible to increase the process execution capacity of the system by employing additional navigator threads.

[6] As explained in Section 7.4.2 on page 167, a service may be invoked immediately, synchronously, or asynchronously. Furthermore, for some component types, before the invocation takes place a suitable provider may have to be chosen.

[7] Each execution subsystem may use its own threading model to handle the synchronous or asynchronous interaction with the actual service provider

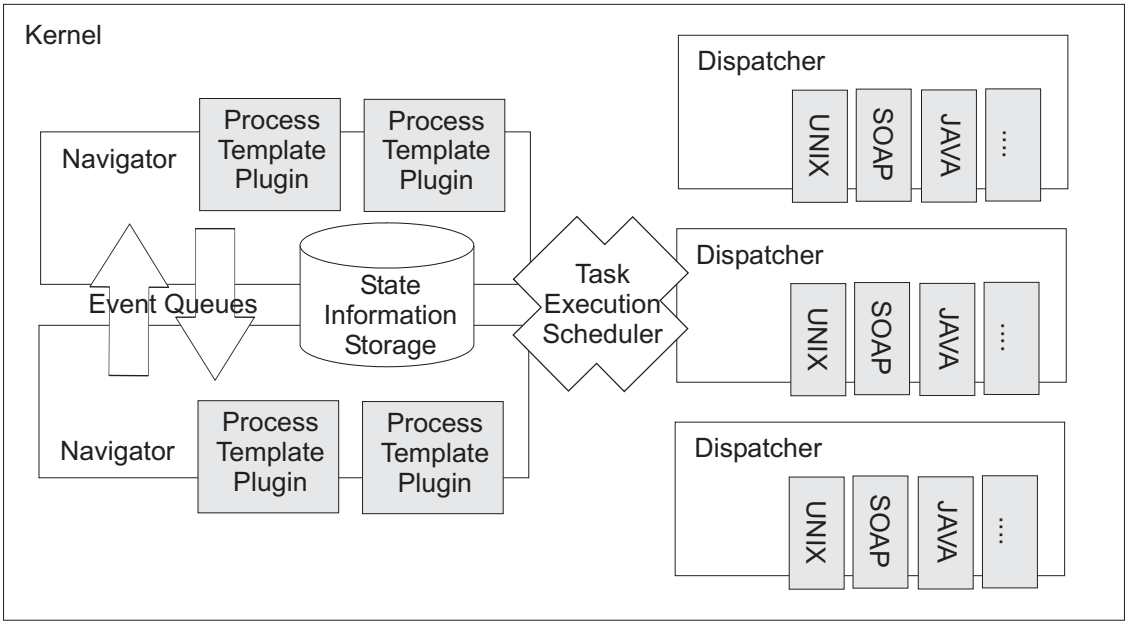


Figure 7.6.: *Architecture of a Distributed Kernel*

- By monitoring the length of the queues which buffer the interaction between navigator and dispatcher threads it is possible to dynamically determine the optimal number of threads on each side automatically, e.g., with the goal of keeping the average queue length constant.
- By extending the event notification, the task execution scheduler, and the state information storage components with transactional properties, it is possible to achieve fault tolerance. This way, all interactions of the navigator thread with the state information storage component during the processing of an event happen atomically. If the navigator fails, the state of both the event notification and storage components is rolled back, as the transaction is aborted. Thus, the system can recover immediately, because a different navigator thread may resume the work of the failed one.

7.3.3. Deployment Scenarios

In this section we present some of the configurations, listed in Table 7.1, in which our flexible kernel architecture can be deployed and discuss the main advantages and disadvantages regarding their performance.

Not only flexibility is an important aspect for performance reasons, but it allows the system to be adapted to different requirements, so that it can be deployed into several environments and configurations to match a specific workload target. For example, our architecture can be deployed as a light-weight process simulation engine, attached to a process development tool. Similarly, it can be embedded into standalone Java applications that require process enactment capabilities to coordinate

the invocation of different components and services. This way, the coordination logic specified as a process can be directly executed within the context of the application. Alternatively, the JOpera kernel can be used as a reliable service orchestration platform running inside an application server (or an enterprise service bus), which can also scale to handle very large workloads, using a cluster based configuration.

Furthermore, flexibility is one of the basic requirement of an infrastructure capable of exhibiting autonomic behaviour [109]. To this end, it is important that the system can be reconfigured dynamically following the decisions of an autonomic system controller, which monitors the current workload conditions and determines the optimal system configuration [21].

The simplest configuration (a) is a so-called *monolithic kernel*, where one navigator and one dispatcher run on the same machine. The state information storage, the event queues and the task execution services are implemented using the appropriate main memory data structures. Since all data is kept in main memory, this configuration trades recoverability from failures with very fast access to the state information. Given its centralized nature, such an architecture doesn't scale well with large workloads, both because there are only a single navigator and one dispatcher components, and because, when managing a very large number of process instances, the system may run out of memory space[8]. In addition to the ease of deployment, its main benefit lies in its very low overhead with small workloads.

The next configuration (b) is called monolithic *persistent* kernel. Again, one navigator and one dispatcher run on the same machine, but the storage of the state information is implemented using a remote, persistent, data repository. This makes the kernel recoverable, at the cost of a larger overhead[9].

The limitations of these centralized configurations concern all five main system components: the Navigator, the Dispatcher, the Task Execution Scheduler, the State Information Storage and the Event Queues. If one is replicated in order to improve its throughput, very soon another component becomes a bottleneck. For example, if a set of navigators send task execution requests to the dispatchers through a centralized scheduler, the throughput of the scheduler limits the rate at which tasks can be executed. Similarly, if the performance of the state information storage improves, the navigator will be able to produce and consume events at a higher rate, putting a higher burden on the event queues. Thus, while scaling up the system and configuring it with replicated components (Figure 7.6), care must be taken to keep the system well balanced.

The first replicated configuration we present concerns the dispatcher component. In this case, a single navigator (with (d) or without (c) persistent storage) manages the processes, whose tasks are executed by an increasingly large number of dispatchers. As the task execution capacity of the system increases, it is to be expected that the system may be capable of handling a larger workload. As the measurements show, this is only true when the task duration is long enough, that is longer than 10

[8] See Figure 8.6 on page 189 for some measurements of the memory consumption of a monolithic kernel.

[9] as the results shown in Figure 8.3 on page 186 indicate.

	Event Queues	State Information Storage	Task Execution Scheduler	Dispatcher	Navigator
(a)	Local	Volatile	Local	Single	Single
(b)	Local	Persistent	Local	Single	Single
(c)	Centralized	Volatile	Remote	Multiple	Single
(d)	Centralized	Persistent	Remote	Multiple	Single
(e)	Distributed	Volatile	Remote	Multiple	Multiple
(f)	Distributed	Persistent	Remote	Multiple	Multiple

Table 7.1.: *Deployment Scenarios*

seconds. For tasks lasting a shorter time, the actual bottleneck lies in the navigator component.

This problem is addressed by the configurations (e) and (f), where also the navigator component is replicated, keeping the number of dispatchers and the corresponding task execution capacity constant. As our measurements indicate, the system's scalability is now bound by the persistent storage service. In fact, using a centralized data repository with an increasingly large number of clients (the navigators) only scales up to a certain limit. Therefore, we also tested a configuration (e) having the storage of the state information localized at the navigator. Because of the improved performance of the storage service, the limiting factor shifted to the event communication service, which also had to be partitioned in order to keep the system functioning.

7.3.4. Parallel Navigation

As we have shown in the previous section, the JOpera Kernel is flexible enough so that it can be deployed in different configurations. One key aspect that enables such flexibility is the execution of the navigation algorithm in parallel on multiple process instances. To do so, we chose to parallelize the navigation algorithm based on the observation that each process instance is a fully independent entity. In particular, changes to the state of one instance do not affect other process instances. Therefore, it is possible to partition the system's workload at the granularity level of the process instance and perform navigation on different, independent process instances in parallel. As a consequence, the navigation algorithm presented here doesn't need to be changed, since it can be implemented in a thread-safe manner. However, the underlying infrastructure needs to support the concurrent execution of the algorithm, triggered by events concerning independent process instances.

Once the system is capable of performing parallel navigation, issues such as load balancing and fault tolerance should be dealt with. In our current prototype we support two different load balancing strategies: either process instances can be statically partitioned among different parallel navigators (load sharing), or events and state information can be dynamically moved between different navigators in order

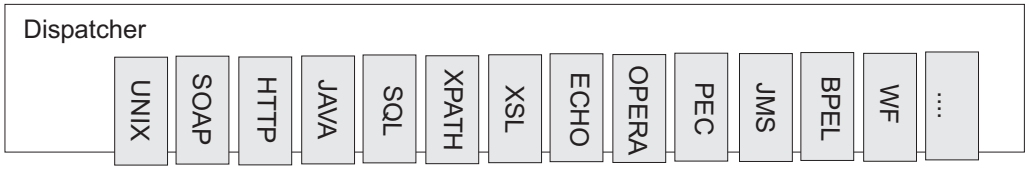


Figure 7.7.: *The dispatcher as a container of task execution subsystem plugins*

to keep the system balanced. Moreover, since each navigational step is executed atomically within a transaction, and if the state information can be stored remotely and persistently, recovery from failures occurring in the navigator becomes completely transparent. In fact, when a dynamic load balancing strategy is used, upon detection of the failure, the process instances belonging to a failed navigator can be immediately assigned to another one.

Another interesting set of scheduling problems stem from the possibility of having an heterogeneous set of available plugins loaded in both the navigator and dispatcher components. As a consequence, the routing of the events to the appropriate navigator on one side, and the scheduling of the task execution requests on the other, become more complex, because not all of the dispatchers (and navigators) have access to the same set of plugins. On the one hand, by using Java code mobility techniques, it is always possible to upload a missing process plugin into a navigator which has received an event for an unknown process. However, it is necessary to devise a mechanism to locate the required process plugin and include a policy to determine in advance, considering the overall workload distribution, the benefit of such migration of the process plugin. On the other hand, the service invocation adapters loaded into the dispatchers are determined by the system configuration, as we will discuss in the next section. Therefore the task execution scheduler should not assume an homogeneous environment.

7.4. Supporting Heterogeneous Services

After presenting the set of components into which JOpera's process execution kernel has been structured, in this section we focus on one of them: the *dispatcher*, with the aim of describing how JOpera's flexible component meta-model (Chapter 4) has been implemented. In order to support an open-ended set of component types – including the ones listed in Table 4.1 on page 51 – flexibility is an important property associated with the design of both the Opera Modeling Language as well as the JOpera system itself.

Flexibility in the design of the system is achieved through a plugin-based architecture, where an abstract interface used by the rest of the system to perform task execution is mapped to the actual service invocation mechanism, which is related to the type of component. This mapping is implemented by an adapter module: the *task execution subsystem*, which can be dynamically plugged into the dispatcher. Thus, the dispatcher acts as a container of task execution subsystem plugins (Fig-

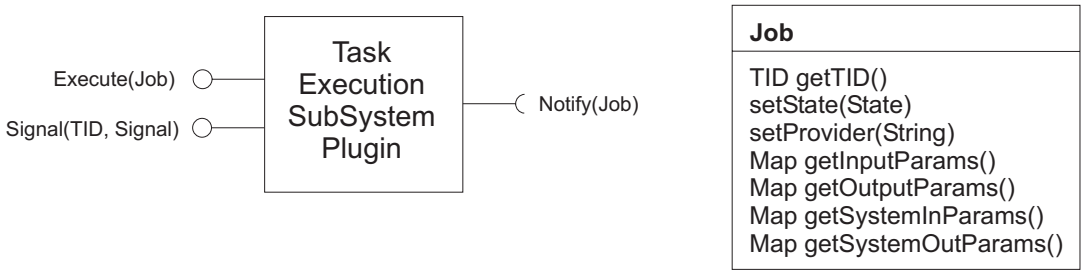


Figure 7.8.: *Interface definition of the task execution subsystem*
The UML component diagram on the left defines the provided and required services of a task execution subsystem. The class diagram on the right defines the attributes of the `Job` interface, which is passed as a parameter to the task execution subsystem plugin.

ure 7.7), factoring out common functionality such as thread pooling, handling of failed service invocations, and dispatching the task execution requests submitted by the navigator to a suitable plugin.

Here is an example to motivate such plugin based approach. The task execution request to run a UNIX command line should be mapped to the corresponding *fork/exec* system calls, while the task execution request to invoke a remote Web service is mapped to the corresponding SOAP message round. Although these two service invocation mechanisms are completely different – the first involves the interaction with the local operating system, while the second requires (in most cases) the exchange of XML documents over an HTTP connection – they can be unified under a common interface, to which any task execution plugin must conform.

As shown in the UML diagram of Figure 7.8, this `SubSystem` interface has been kept by design very simple, shifting the complexity of different types of interaction patterns and service invocation mechanisms into the `Job` parameter. A task execution plugin must implement at least the `Execute` method, while the `Signal` method is optional, as not all service invocation protocols support the signaling of an active service invocation.

Similar to the issues discussed when defining JOpera's component meta-model, behind the design of such an interface there are the following considerations:

- *Control flow.* Through the same interface different patterns of interactions should be supported, as far as the control flow is concerned. This implies that a mechanism is required for using the synchronous Java method call of the `Execute` method – which initiates the service invocation through the plugin – to support asynchronous interactions, where the invocation of the service does not complete when the call of the `Execute` method returns. In case of asynchronous invocations, the plugin is responsible of sending a notification event through the `Notify` method, which exposes to the plugin the event queue component of the kernel.

  In order to support additional interactions with a service invocation, such as

the possibility of interrupting it, the plugin has the `Signal` method, which takes both the task identifier (`TID`) and the required `Signal` as parameters. As opposed to having a method for each possible interaction, to ensure the flexibility of the resulting system, the `Signal` parameter is used to specify the required interaction. Its value is interpreted by the plugin, and the set of possible values (currently: `Abort`, `Suspend`, `Resume`) is not restricted *apriori*. This way, as long as existing plugins validate their input, it becomes possible to extend JOpera with additional interaction capabilities, without breaking the task execution subsystem interface.

- *Data flow.* All plugins must conform to JOpera's meta-model based on system parameters. In other words, the notion of system input and output parameters is shared among all plugin implementations. However, the actual set of system parameters understood and expected by a certain plugin depends on the corresponding service invocation mechanism, as defined in Chapter 4.

  Furthermore, regarding output parameters, the mapping between system and user defined parameters is also performed by the task execution plugin. However, within JOpera there is a set of predefined mappings, which are thus available to be reused among all plugin implementations.

- *Failures.* It is the responsibility of the plugin to determine whether the service invocation has succeeded or not, and to map the failure modes of the specific protocol back to JOpera's system parameters. As far as the plugin interface is concerned, Failures are not signaled through Java exceptions. Instead, error conditions are set as part of the `Job` parameter because – as previously mentioned – the outcome of a service invocation may not always be known when the `Execute` method returns.

As a final remark, in order to facilitate the integration of adapters for many different types of components, the simplicity of the interface's contract is a very important practical aspect. If it would turn out to be too difficult to fulfill the contract behind the task execution interface, the number of available task execution plugins would not be very large and the viability of the component meta-model would suffer.

7.4.1. Describing a task execution request

As shown in Figure 7.8, the information describing a task execution request has been structured into the `Job` class. Although some of the attributes (e.g., the `TID` used to identify the job) are shared among all types of service invocations, a task execution request also contains information which is strictly related to the mechanism used to invoke the service.

The choice of how such information is described as it flows through the various layers of the system (i.e., from the process model at design-time to the service invocation at run-time) constitutes an important design decision, which affects both the efficiency and the flexibility of the system (Table 7.2).

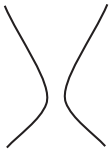
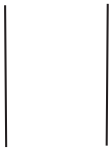
System Layer	(1)	(2)	(3)
User (JVCL) Process Model (OML) Task Execution Subsystem Service Provider			
Flexibility	high	high	low
Performance	low	medium	high
Coupling	loose	medium	tight

Table 7.2.: *Design options to describe a task execution request.*
The lines represent the degree of coupling between the various layers of the system.

1. On one side of the spectrum, at runtime, all information concerning a task execution request is encoded as a string [21]. This solution emphasizes flexibility at the expense of performance. On the one hand, the interface of all task execution plugins is simplified, as it only receives a *command* string. On the other hand, as the task execution request is prepared, all related information (e.g., values of parameters) must be encoded in such string, which must be later decoded to be interpreted by the task execution subsystem plugin.

2. As a compromise, in order to store separately different sets of parameters, it is possible to use an associative data structure (called `Map` in Java) to store a set of key-value pairs. As shown in Figure 7.8, this is the current approach in JOpera, as it allows to keep a generic (and, most important, open) plugin interface (based on the `Job` class) and avoids the runtime costs due to the encoding and the decoding of command strings of the previous solution.

3. At the other end of the possible solutions, a viable design in case of a system with a fixed (and closed) set of component types would be as follows. For each component type, the information describing a task execution request would be stored into a different Java class (implementing the `Job` interface), whose fields would correspond to the system parameters listed in the component type description. At runtime, the task execution request would be stored in an object of one of such classes by the navigator's process plugin. The object would be thus transferred from the navigator to the task execution subsystem without any additional overhead. This way, as depicted in Table 7.2, the interface of the task execution subsystem would not constitute a bottleneck in the flow of information between different parts of the kernel.

These different solutions also have an impact on the possibility of checking that the jobs are correctly passed to the appropriate task execution plugin. Although with the first two solutions it becomes easy to fit a new plugin within the JOpera system, the possibility of statically checking that the plugin receives only matching

execution requests is greatly reduced, as neither the compiler nor JOpera can verify the format of the command string or the content of the parameter maps. In reverse, the last option would also allow to use type checking to ensure that jobs are routed to the appropriate plugin. However, in this case the programming effort required when developing a new plugin would significantly increase as opposed to the previous solutions. In all cases, regardless of the way the description of the service invocation is encoded, it still remains the responsibility of the task execution plugins to ensure the correctness of the values of the system parameters they receive, in order to catch some possible error conditions before the service invocations are attempted.

7.4.2. Dispatching a task execution request

Once a task execution request has been submitted by the navigator thread, as it has been discussed in Section 7.3.2 on page 157, the dispatcher thread is responsible for performing the corresponding service invocation. As this happens, depending on the actual type of service to be invoked, a task execution request (or a *job*) goes through several stages in order to support late binding, as well as for providing a certain degree of fault tolerance.

Before the actual service invocation can be carried out, given the requirements associated with the job, a suitable task execution subsystem must be found. This way, only at the latest possible moment the system chooses which access method should be used to contact the service provider. If a matching subsystem plugin cannot be located, the task execution fails.

Furthermore, if during the invocation an error occurs, which prevents the completion of the invocation, the dispatcher attempts to locate an alternative access method to automatically retry the invocation up to a certain number of times. As opposed to failures detected after the invocation of a service has completed, i.e., the results returned by the service invocation are incorrect, this error condition corresponds to the impossibility of initiating a service invocation, due to, e.g., the unavailability of a service provider. To keep this retry-based fault tolerance mechanism independent of the task execution subsystem, the `Execute` method raises an exception to signal that such an error has occurred and that the service invocation should be retried.

Especially when dealing with unresponsive service providers, where the invocation of a service fails because an answer from the corresponding provider doesn't come after a certain timeout has expired, it is important to guarantee that a slow service invocation does not affect, nor delays other independent ones running concurrently. Therefore, to dispatch task execution requests, JOpera uses a pool of threads of a configurable size. The previous description still holds, as this is corresponds of having a pool of one thread. However, in more realistic setting, to provide enough task execution capacity, the *logical* dispatcher thread corresponds to a *physical* pool of threads, whose size determines the maximum number of tasks that the system can execute in parallel. As we have previously mentioned, depending on JOpera's deployment configuration, these threads can be distributed across several machines.

Another approach to reduce the number of threads that are used for executing

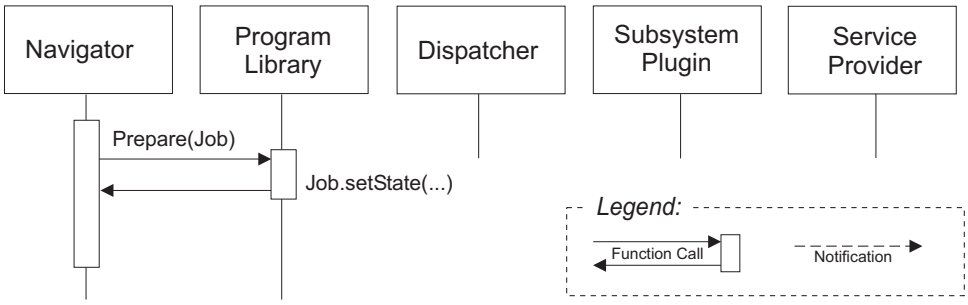


Figure 7.9.: *Immediate service invocation*

service invocations can be taken within the implementation of a task execution subsystem. If applicable to the specifics of the service invocation protocol involved, a thread may be used only to initiate the service invocation and, as opposed to blocking its execution in order to wait for the response of the service provider, the rest of the invocation may be carried out asynchronously. Therefore, the dispatcher thread can be returned to the pool, so that it can be used to execute other jobs. However, in order for this optimization to work, the interaction between the subsystem and the service provider must be asynchronous, i.e., the response of the service provider has to come through a separate channel and the subsystem itself should implement the listener required to correlate the response messages to the pending jobs, and to send the notification event back to the navigator. Considering the component types listed in Chapter 4, there are many candidates for which this optimization may be applied. Not only messaging components and job submissions to cluster computing environments, but workflow tasks, and sub-process calls are also implemented this way.

7.4.3. Service invocation patterns

Proceeding from the previous observations, in the design of the JOpera kernel we distinguish between different patterns of interaction between the various kernel components and the service provider, in order to perform a service invocation in the most appropriate way.

Depending on its component type, a service can be invoked immediately (Figure 7.9), synchronously (Figure 7.10), or asynchronously (Figure 7.11). Furthermore, in case it is possible to choose between multiple providers, such synchronous (Figure 7.12) and asynchronous (Figure 7.13) invocations can be scheduled, i.e., the provider to be contacted is chosen among the set of available ones before the invocation takes place.

Immediate service invocation

This service invocation patterns offers the most efficient way of interacting with a service, as the invocation is implemented as a method call executed within the

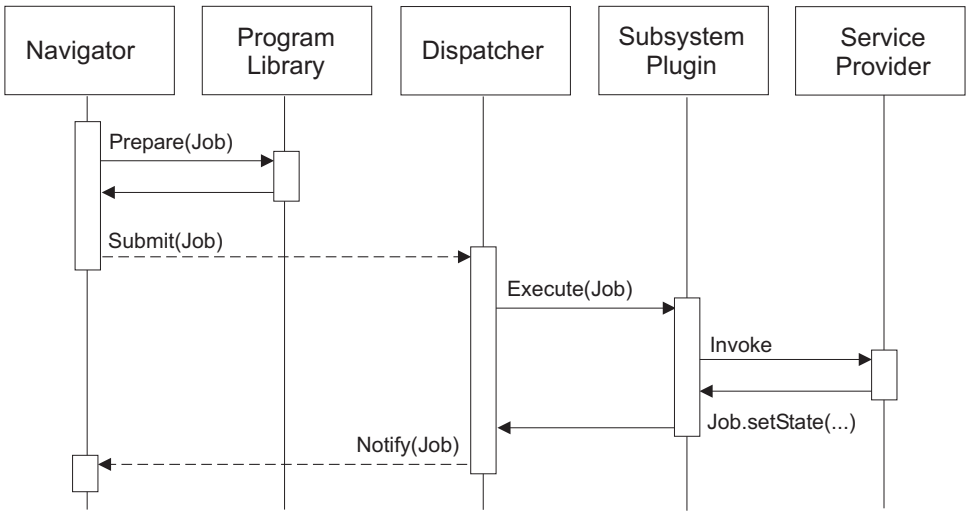


Figure 7.10.: *Synchronous service invocation*

context of the navigator thread. As shown in Figure 7.9, once it has determined that a certain task should be executed, the Navigator contacts the Program Library to prepare the corresponding job (or task execution request). In case of programs corresponding to the "Java Script" (Section 4.5.1) component type, the code entered by the developer has been embedded by the compiler in the job preparation code. This way, it can be *immediately* executed.

Furthermore, the compiler inserts a call to the `Job.setState(...)` method after the embedded Java code. By checking the state of the Job after preparing it, the navigator can detect that a job has not only been prepared but it has already finished (or failed), thus it is not necessary to submit it to the task execution scheduler, and navigation over the rest of the process can continue without delay.

One limitation of this interaction pattern is that the execution of the service is carried out within the same thread responsible for executing process navigation. Therefore, the benefit of reducing the overhead for one service invocation must be discounted with a reduction in the throughput of the navigator thread. The duration of the computations performed by services invoked following this pattern must be kept small, in order to reduce their impact on the overall performance of the system.

Synchronous service invocation

After a job has been prepared by the navigator thread, in case it still needs to be executed – as in most cases – the job is submitted through the task execution scheduler to the dispatcher thread (Figure 7.10). As previously described, the dispatcher locates the most appropriate task execution subsystem and forwards to it the job by calling its `Execute` method, which maps the task execution request to the actual mechanisms and protocols corresponding to the component type.

In case of component types which support the synchronous interaction with the service provider – e.g., a Web service contacted through the RPC style (Sec-

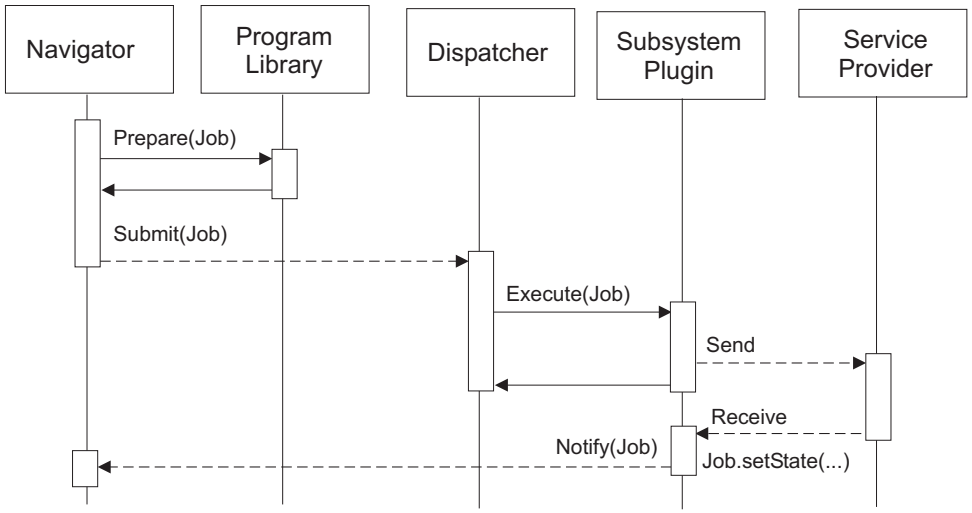


Figure 7.11.: *Asynchronous service invocation*

tion 4.3.1), the invocation of shell command (Section 4.4), or the execution of an XML transformation (Section 4.7.1) – the invocation of the service happens through a method call. The completion of such call indicates that the service invocation has finished (or failed). Thus, the subsystem can update the state of the job with the results of the call before its `Execute` method terminates. The dispatcher detects that the job has already been completed and notifies the navigator thread of such event.

Asynchronous service invocation

Considering component types that involve the asynchronous interaction with a service provider, e.g., through message exchange or non blocking calls, the implementation of the `Execute` method of the subsystem plugin changes, as shown in Figure 7.11. In it, the service invocation is only initiated and the state of the job is left untouched. This way, the dispatcher can catch this condition and avoid sending any notification back to the navigator, as the task execution has not yet completed.

Once this happen, the service provider contacts the subsystem plugin through the appropriate mechanism dependent on the component type to send it the results of the service invocation. The plugin can then extract them from the message and perform the inverse mapping from system parameters to user defined parameters. Finally, an event with such results, formatted in a way that can be understood by the rest of the JOpera system, is sent to the navigator, to indicate that the task execution has completed.

Scheduled service invocation

The difference between synchronous and asynchronous service invocation is shown in the UML sequence diagram of Figures 7.10 and 7.11. For both of these patterns

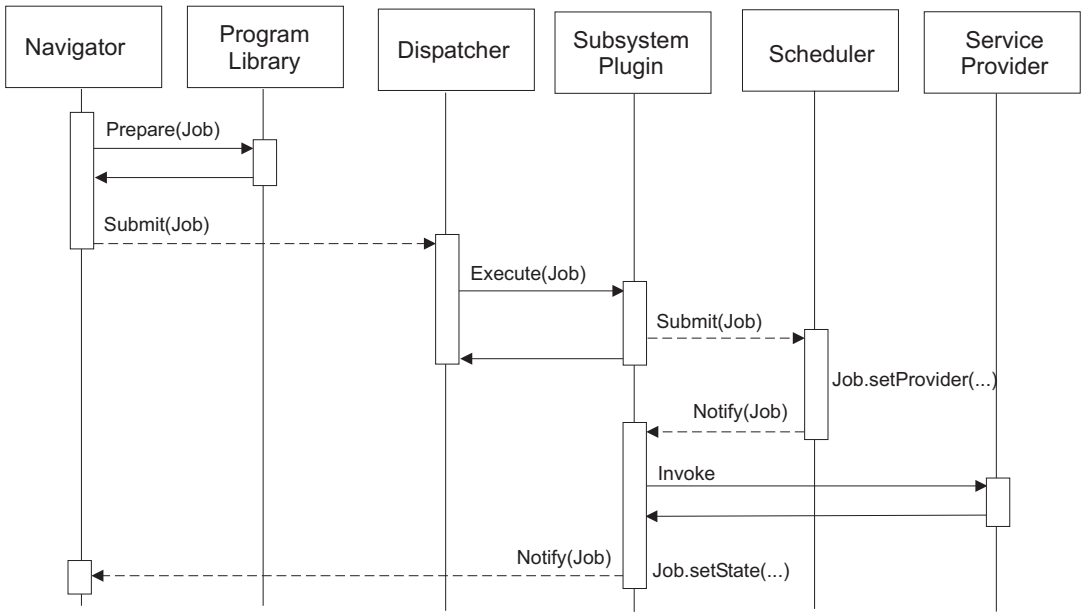


Figure 7.12.: *Scheduled synchronous service invocation*

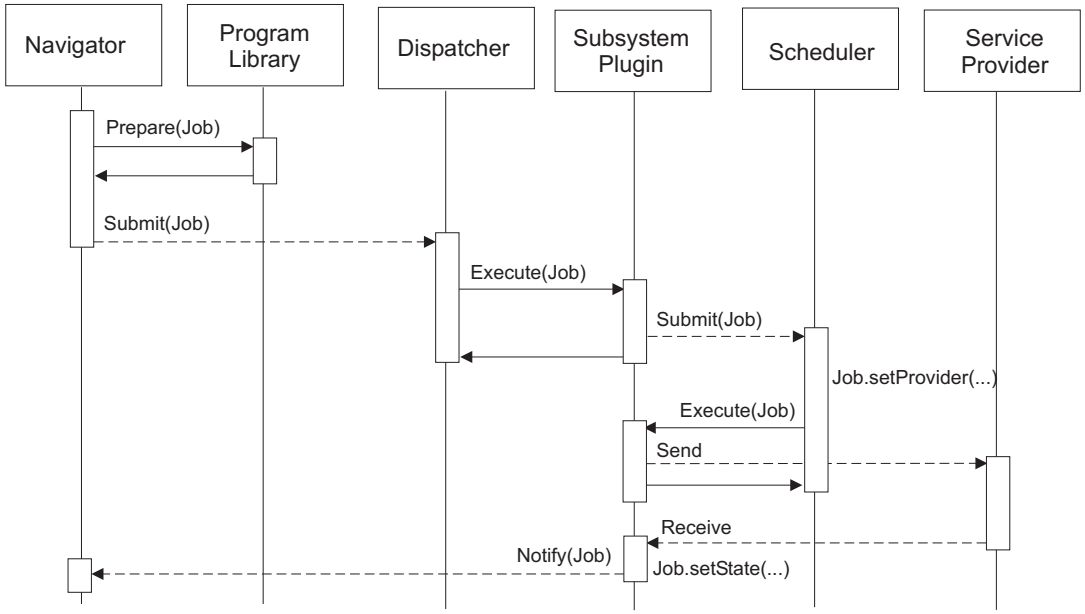


Figure 7.13.: *Scheduled asynchronous service invocation*

it is possible to introduce an additional, intermediate scheduling step, where the service provider to be invoked is chosen among a set of alternative ones.

To do so, an additional component is introduced, which provides scheduling services to the task execution subsystem plugins. In some cases, e.g., where the submission of a job to be executed on a cluster of computers is involved, the scheduling component is external. In others, when choosing the optimal database server to which a certain SQL statement should be submitted, the scheduler is internal to the plugin. Likewise, the choice of the most appropriate human operator to which a certain workflow task should be assigned is performed within the worklist handler subsystem.

In all cases, before a service can be invoked, a suitable provider for it must be located. Therefore, the `Execute` method of the plugins supporting this feature is divided in two parts. The first part involves the submission of a scheduling request, while the second one involves the actual synchronous (or asynchronous) service invocation, which remains very similar to the previous description.

The scheduling policies [213] that are applicable to the choice of the provider also depend on the component type involved. For example, the information policy specifies what load information is available to the scheduler and when and how that information is sent to the scheduler (e.g., on demand, periodically, or upon significant change). In case of a cluster of computers managed by BioOpera, each node of the cluster can be considered a different provider of computational services [23]. The load of such a node can be easily defined in terms of CPU utilization and available memory. For other types of components (e.g., a remote Web server) it may be more difficult or even impossible to perform measurements of the load of a certain provider. Thus, the *feedback* from the provider that could be used to make more informed scheduling decisions is missing. However, for a given service provider, it is always possible to track its availability state and the number of outstanding service invocations, so that – at least in *feedforward* – the scheduling control loop can be closed.

7.5. API

In JOpera's architecture, the compiler links the design tools to the process execution kernel. At runtime, the monitoring, debugging, and system administration tools connect to the process execution kernel through its Application Programming Interface (API). Additionally, the functionality of this API is also available to be called within JOpera processes, through reflection based on system services[10]. This API can be structured in two different parts: the Process Control API and the Program Library Management API.

[10]See Section 3.7 on page 34 for examples on how to call JOpera's API from within a JOpera process.

7.5.1. Process Control

In the following we describe in detail the most relevant functions of the Process Control API, also listed in Table 7.3.

API Function	Description
`login`	Authenticate with the system.
`start`	Start a new process instance.
`query_state`	Query about the state of an existing instance.
`query_data`	Query about the parameter values of an instance.
`set_data`	Update the value of a parameter value of an instance.
`query_interface`	Query about the interface definition of a process template.
`list_instances`	Return the list of all instances.
`list_templates`	Return the list of all templates.
`signal`	Interact with an instance: kill, restart, suspend, or resume it.
`delete`	Delete an inactive instance.
`subscribe`	Subscribe to the state change events of an instance.
`unsubscribe`	Unsubscribe to the events generated by an instance.

Table 7.3.: *Summary of the JOpera Process Control API*

Login

The `login(user, password)` function authenticates a user with the system and, if successful, returns a security session token which must be passed to every other function of the API. The password is sent as a hash value [204]. The system may also be configured to communicate this sensitive information internally using SSL and externally using SOAP over HTTPS. The life cycle of the session is maintained using a soft state technique [194], i.e., if no activity is detected after a configurable timeout the session's validity expires and the user must use the `login` function again.

Start

The `start(process, parameters, options)` function is used to create a new instance for a process template. The caller may pass parameters which will be assigned to the input parameters of the new instance. If no error occurred, the function returns the `id` identifying the new instance.

In addition to input parameters it is also possible to specify some options to control how the new instance will be managed by the system. In addition to the system parameters presented in Section 4.8.2 on page 74, through this API function is possible to access the following two additional options: First of all, with the `subscribe` option, it is also possible to atomically start a process and subscribe

to it, for example, to be notified when it has finished. There is also an optional flags indicating whether the process instance should be automatically deleted upon successful termination (`delete_on_finish`), or conversely it should be kept in the history database so that its results can be reused by other instances.

Query Instance State and Data

The `query_state(id)` function returns the current state of a certain process instance. State information includes whether a process is running, it has finished or it has failed, and timing information measuring, for example, how long the execution of the process took.

Similarly, the `query_data(id)` function returns the above state information plus the current content of all input and output parameters of a certain instance identified by its `id`. For debugging purposes, it is possible to use the `set_data(id,value)` function to update the value of a parameter of a certain instance. In order to ensure the consistency of the process, this action should be only performed on paused process instances.

Query Template Interface

The `query_interface(process)` function returns the list of input and output parameters describing the interface of a certain process. This information, as with most of the other API functions, is returned formatted in XML. This function was originally intended to be used before starting a process, in order to present the user with an input form where she may select or enter the values of the input parameters of the new instance. It turned out to be very useful when implementing the automatic translation to WSDL of the interface of a process published as a Web service.

List Templates and Instances

The `list_templates` function returns the list of all process templates registered within the system. This list contains some metadata about the templates, such as their id, name, author, description, and flags indicating whether the template should be made visible as a Web service or not and whether it can be started by the user. For a given process template, the function `list_instances(process, filter)` returns the list of its instances. Since the resulting list may be quite large, it is possible to select a `filter` criteria to make JOpera only return the relevant instances, for example, to list only the processes that have failed or have been aborted, the ones started by a particular user in a certain time range, or the ones having a specified set of input parameter values.

Signal and Delete an Instance

The `signal(id, signal)` and `delete(id)` functions are used to interact with instances identified by the specified `id`. Possible signals for an instance are `kill` to

abort its execution, `restart`, `suspend` to pause its execution, and `resume` to continue executing a suspended instance. Instances can only be deleted when they are inactive; attempting to delete a running instance will result in an error.

Event Notification

The `subscribe(id)` and `unsubscribe(id)` function is used to register and deregister a listener for state change events related to a certain process instance. The `subscribe` function is symmetric to the `query_state` function, in the sense that it lets the client be notified as soon as a state change occurs, instead of having the client polling the system about the state of a particular instance with a certain frequency. The notification of an event to the registered listeners occurs using an internal protocol.

7.5.2. Program Library Management

The first set of functions of the program library management API listed in Table 7.4 are used to query JOpera's registry of available programs. The second ones allows developers to register new programs and modify the access methods of existing ones.

Query and List

The query and list functions are used to retrieve lists of programs which match a certain criteria.

The `query_program(name)` looks up in JOpera's registry for programs with a matching `name` and returns all information relative to one (i.e. its interface and its access methods). Additionally, the `list_interface(id)` functions searches the program library for programs which are compatible with a given interface. Such interface is identified by the `id` of the task which uses it. To control the number of returned matches, it is possible to specify with an additional, optional parameter, whether one or more results should be given. If no matches can be found, the invocation of the API function fails.As we have shown in Example 3.2 on page 36 these functions can be used to support the dynamic choice of an implementation to match a given service interface. In case a list of more than one matching programs is found, in this scenario, it is necessary to process the list to choose the optimal service implementation.

The `list_dependency(name)` is used to determine, given a program's `name`, which are the processes currently referring to it. With this function, system administrators can determine the impact of a modification to a program, i.e., by listing which are the processes that will be affected if a program is changed. The `list_programs(filter)` returns the list of all matching programs. In the simplest case, all programs in the library are returned. Otherwise it is possible to apply several filtering criteria. For example, to obtain the list of programs used within a given process. The keys returned by this function should be passed to the `query_program` to obtain specific information about the individual program.

API Function	Description
`query_program`	Lookup a program based on its name.
`list_interface`	Lookup a program based on an interface.
`list_dependency`	Return all processes that refer to a certain program.
`list_programs`	Return the list of all matching programs.
`update_program`	Modify the access methods of a program.
`delete_program`	Remove a program definition.

Table 7.4.: *Summary of the JOpera Program Library Management API*

Update

The `update_program(oml)` is used both to insert a new program registration in the library and to update the information about an existing one. In the latter case, only changes to its access methods are allowed, while the interface cannot be modified, as this would invalidate the references of the activities which call the updated program. Given the clear separation between the processes and the programs, which are linked by the interface, changes to the internal details of a program do not affect the processes using it, i.e. such processes do not have to be recompiled. Furthermore, after the modification of a program has been stored, the updated information is immediately available to the rest of the system, and is used both by new instances and processes that were already active before the update was made.

The `delete_program(name)` removes a program registration from the library.

7.6. Discussion

As we have mentioned at the beginning of this chapter, the design of JOpera's architecture has been influenced by two contrasting requirements: efficiency and flexibility. Flexibility is very important in order to support the invocation of services belonging to an open and rich set of different component types (Section 7.4), as well as to build a system that can be easily configured to be deployed in different settings. This way, as described in Section 7.3.3, the configuration of the system can be tailored to scale from a light-weight, embedded process simulation kernel to a cluster-based, distributed process execution engine. Flexibility, however, comes with a price. In particular, there is a trade off between the efficiency of the system and its flexibility.

To address this trade-off, in this chapter we leveraged the flexibility of the architecture to be able to include or exclude features which may constitute a potential efficiency bottleneck. For example, if JOpera should be used as a process development platform for rapidly building and testing processes composed of different services, then the expensive reliability features (involving a transactional and persistent implementation of the state information storage) are not required. Therefore, to suit this usage scenario, JOpera can be deployed without them.

Likewise, the flexibility which allows to replace the implementation of the state

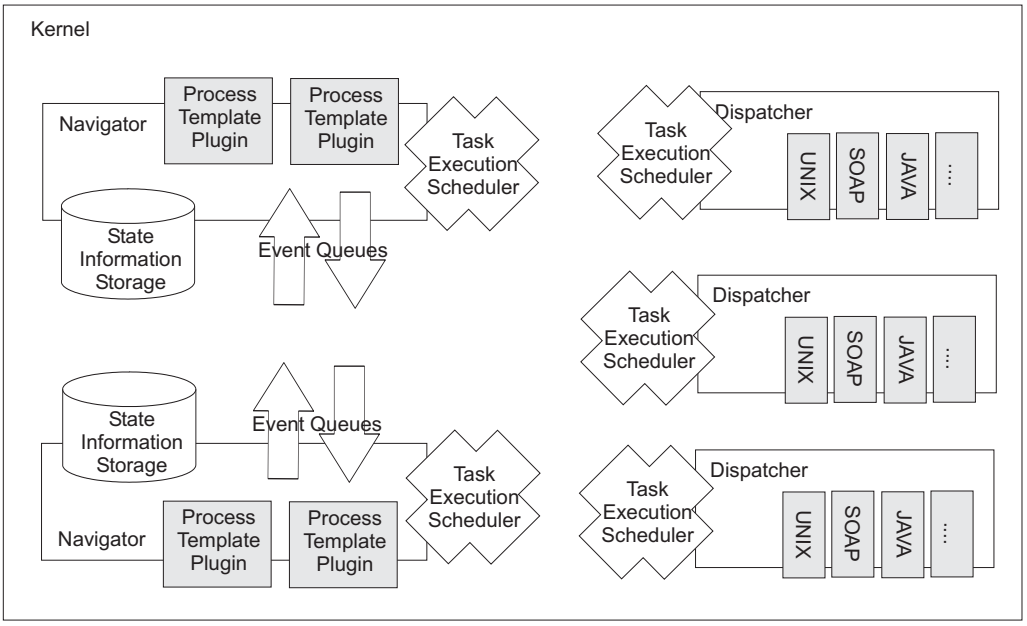


Figure 7.14.: *Architecture of a Peer to Peer Process Execution Kernel*

information storage, event queue and task execution scheduler components without affecting the rest of the system (and most important, the compiler) can be exploited to experiment with alternative infrastructures. For example, the state information storage can be optimized for different database management systems, as long as its interface to the rest of the system isn't modified, so that it is possible to leverage existing database installations when deploying the kernel. If an existing database is not available, it is always possible, for example, to replace the implementation of the storage subsystem with a file-based one.

Another possible extension of these basic services consists of using emerging peer to peer technologies in order to implement a highly dynamic system where both the state of the processes and navigational events are propagated without relying on a centralized component. This way, it becomes possible to produce a peer to peer version of the kernel, in which the execution of the processes is shared among several peers through a server-less implementation of the state information storage, event and task scheduling subsystems (Figure 7.14). Currently, it remains an interesting, open question, whether existing peer to peer technologies can provide a reasonable level of performance, both in terms of reliability and scalability, when applied to this application scenario. Still, we believe that the architecture of the system – as we have presented it in this chapter – would not radically change, although the system could become the basis for a new type of process-based distributed computing environment. In it, the current asymmetry of peers contributing cycles to existing computations and problems without the opportunity of submitting and starting their own ones would cease to exist, because we speculate that all peers would share the ability of both executing and starting processes. Therefore, the reward to motivate

new peers to join a distributed computation would lie in the ability of every peer running a JOpera kernel to compile, submit and monitor the progress of his or her own process, which would be run using shared resources of potentially many other peers.

A flexibile architecture is also important in terms of providing dynamic, automatic system reconfiguration. To do so, JOpera's infrastucture provides the basic mechanisms to grow, shrink or migrate parts of the system, e.g., by starting or stopping navigator and dispatcher threads across the nodes of a cluster. Currently, these operations are initiated manually, i.e., a system administrator doesn't have to shut down a distributed JOpera kernel to start a new navigator or a dispatcher thread on a new machine. However, as it has been investigated by [21], also with JOpera's flexible architecture it should be possible to include an autonomic controller component, which would periodically monitor the state of a distributed kernel and perform the necessary reconfiguration actions to achieve different goals. These may involve the optimization of resource utilization, the increase of the system's fault tolerance, or the minimization of the turn-around time of the execution of a process instance. Moreover, the autonomic controller would have to be based on a model of the system, which could be used to determine the most useful system parameters (for example, the length of the event and task scheduling queues) that have to be measured to make predicitions about the effect of configuration changes to the behaviour of the system.

Furthermore, flexibility in terms of supporting many different service invocation patterns (Section 7.4.2) is also useful in applying the most efficient one, which is – at the same time – appropriate to interact with the component type in question. For example, immediate invocation (with the performance penality equivalent to a Java method call) is available for minimizing the overhead of invoking fine-grained services. For other component types of a larger granularity, both synchronous and asynchronous interaction patterns are supported, again, in order to enable the development of task execution plugins, for which a perfomance penalty must not be paid due to the inflexibility of the architecture. This argument can also be seen from the opposite perspective. If all service invocations would have to follow the same interaction pattern (assuming that it would be feasible to integrate all types of components) it would not be possible to minimize the service invocation overhead by adapting the service invocation mechanism to the peculiarities of each component type, because of the constraints of such an inflexible architecture. For example, considering the scenario where all service invocations would have to be submitted to a scheduler, even if no multiple, alternative providers are possible. Consequently, for some component types, the scheduling step would not be necessary and thus the invocation of the corresponding services would incur in unnecessary overhead. However, due to the rigidity of the architecture, it would not be possible to bypass such scheduling step, which would also become an important scalability bottleneck.

To what degree we have been successful in designing this flexible architecture without sacrificing the efficiency of the implementation will be quantified in the next chapter with a series of experimental results.

8. Measurements

This chapter presents some experimental results which both motivate and validate JOpera's architecture.

We begin by comparing the service invocation overhead of different component types. This way, we show that, also from a performance perspective, it is important to support the composition of more than just Web services.

Along this direction, we study the performance of a mapping between mismatching service interfaces. This mapping (described in Example 4.3 on page 69) has been both defined visually using the JVCL and textually with an XSLT stylesheet. The performance of the two solutions is compared in Section 8.2.

Finally, in Section 8.3 we present some experimental results concerning the reliability and scalability of different configurations of JOpera's kernel which give an empirical validation of the design decisions that were discussed in the previous chapter.

8.1. Service Invocation Overheads

As discussed in Chapter 4, performance is one of the arguments behind the idea of providing support for invoking services of different component types. In order to give an indication of the overhead involved, we measured the execution time of processes containing only one task. For each process, the task invokes a service of a different type. This way, it is possible to compare the time it takes to invoke a remote Web service across the Internet with the time it takes to perform a local Java method call, and – quantitatively – determine the cost (or the benefit) of preferring services of a certain type over another.

More precisely, in this experiment we compare different access mechanism to the same "Temperature Conversion Service". We choose this service due to its trivial implementation, so that the execution cost is negligible when compared to the overhead of invoking it. Another reason to choose this service is that we found a remote implementation on the Internet at [120]. With it, it becomes possible to present an interesting comparison between the invocation overhead of local and remote Web services.

As listed in Table 8.1, in this performance comparison we use services of various component types and several implementations of the corresponding execution subsystem plugins.

Before presenting the results of the experiment, we make explicit some of the

Component type	Description	Reference
JS	Java Script	Section 4.5.1
OPERA	JOpera SubProcess Call	Section 4.8.2
JAVA	Java Method Call	Section 4.5.2
MSG	Local Message Queue	Section 4.10
JVM	Java Virtual Machine	Section 4.5.4
PYTHON	Python Scripting Language	Section 4.6
SOAP/A11	Local Web Service using Axis 1.1 [12].	Section 4.3
SOAP/A12	Local Web Service using Axis 1.2α.	Section 4.3
SOAP/WS	Remote Web Service using Axis 1.1.	Section 4.3

Table 8.1.: *Service Invocation Mechanisms to be compared*

assumptions about JOpera's configuration. First of all, in this experiment JOpera was running with Java JDK 1.4 on Windows XP, using a Pentium 4 (3Ghz, with Hyperthreading enabled), with 1GB of RAM. To minimize the impact of external factors, we deployed JOpera's kernel in a monolithic, volatile configuration[1]. This way, we ensured that the internal process execution overhead is minimized and the measurements only show the cost of invoking the services using different mechanisms. Furthermore, although several measurements were performed for each component type, at most one service invocation was running at a time. This way, the potential influence of multiple process instances running concurrently was removed. Finally, all plugins (processes and subsystems) have been preloaded into the kernel before the beginning of the experiment.

8.1.1. Results

As shown in Figure 8.1, the most important result of this experiment is that the average service invocation overhead varies about three orders of magnitude (from less than 1 millisecond to 2.31 seconds) depending on the component type.

By further looking at the logarithmic graph in Figure 8.1 (right), it is possible to group the component types in different categories, depending on their overhead.

1. The first four components (JS, OPERA, JAVA, MSG) offer an invocation overhead of significantly less than $1/10^{th}$ of a second, as the implementation of the service is located within the same Java virtual machine where the JOpera kernel is running.

2. Invoking both the JVM and the PYTHON component types requires to spawn a child process in which the corresponding interpreter is started, and this requires more time: about 0.28 seconds. The overhead of this kind of local

[1]Please refer also to Section 7.3.3 on page 160 for more information on this deployment setting.

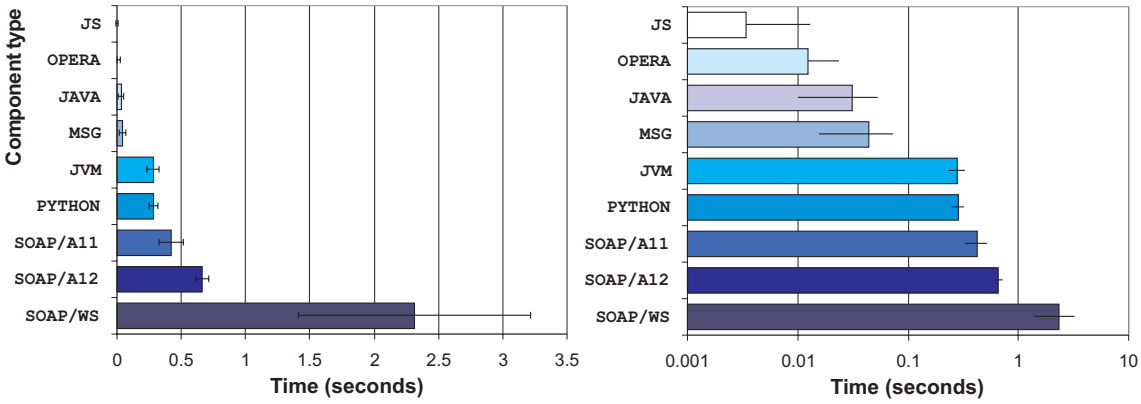


Figure 8.1.: *Service Invocation Overhead for different component types*

inter-process communication is more than 50% of the fastest Web service invocation.

3. The average Web service invocation time is 0.42 seconds in case of a Web service deployed on the local area network, called using Axis version 1.1 (`SOAP/A11`). This time grows to 0.66 seconds using the latest version of Axis 1.2α (`SOAP/A12`). In case of a remote Web service (`SOAP/WS`), the delay and jitter of the wide area network need to be discounted. This effect can be recognized both in the higher (2.31 seconds) average response time and in the very high standard deviation (0.9 seconds).

8.1.2. Discussion

The great variability of the results shown in Figure 8.1 can be partially correlated to the architecture of JOpera's kernel[2].

- In the first case (`JS`) the temperature conversion formula – written as a Java expression – is directly embedded into a process. This component type is thus invoked *immediately*, i.e., within the same navigator thread which is executing the process. Therefore, the very small overhead can be explained by the fact JOpera invokes this service at a cost comparable to the one of a Java method call.

- The `OPERA` component type represents a sub-process call to the previous example, in which the service is implemented as a Java script. The difference in the overhead with the `JS` component type shows the cost of performing such process invocation (about 9 ms with a system running only one process at a time).

[2]In particular refer to Section 7.4 on page 163 for a description of the plug-in based task execution adapters that are used to map the service invocation to the required protocols and mechanisms corresponding to the involved component type.

- The `JAVA` component type represents the synchronous invocation of a method of a Java object, which is created by the Java local method invocation plugin. As opposed to the `JS` component type, in this case the Java code is invoked within a different thread belonging to the dispatcher pool. The difference in the overhead gives an indication of the cost of such context switch.

- The `MSG` component type represents the message-based interaction of two processes. As opposed to a synchronous process call, this example shows the cost of using message queues to control the interaction of two processes. As shown in Example 4.5 on page 81 the client process posts a request with the value of the temperature to be converted in the input queue of the server process, which retrieves it and converts it using the previously mentioned Java expression. The result is returned through the output queue. In this case, the overhead of such interaction is kept very small by the fact that the message queues are implemented locally within the task execution subsystem. Furthermore, such volatile queues do not offer any of the persistence and message routing feature, e.g., of a full JMS implementation [220].

- The invocation of the `JVM` component type involves the execution of an external Java Virtual Machine, which loads the previously mentioned `JAVA` class and wraps it so that the input data can be received from the command line and the results of the temperature conversion can be returned to JOpera's plugin through the standard output pipe. All of these additional operations cause the service invocation using this mechanism to be one order of magnitude more expensive. The same considerations can be applied to the `PYTHON` component type.

- As expected, Web services are the most expensive component type in terms of the overhead involved. Given the current state of flux of the relevant standards and available implementations, the performance of the service invocation may be significantly affected by the choice of which libraries are used. Additionally, the location of the Web service also affects the overhead, as the cost of invoking the remote Web service shows.

  However, it should not be left implicit that this additional cost is due to the distributed nature of the service interaction, in which JOpera and the service provider are separated by the Internet. Thus, it should not be blamed on the Web services protocols, which – instead – are one of the few technologies currently enabling such type of distributed interaction.

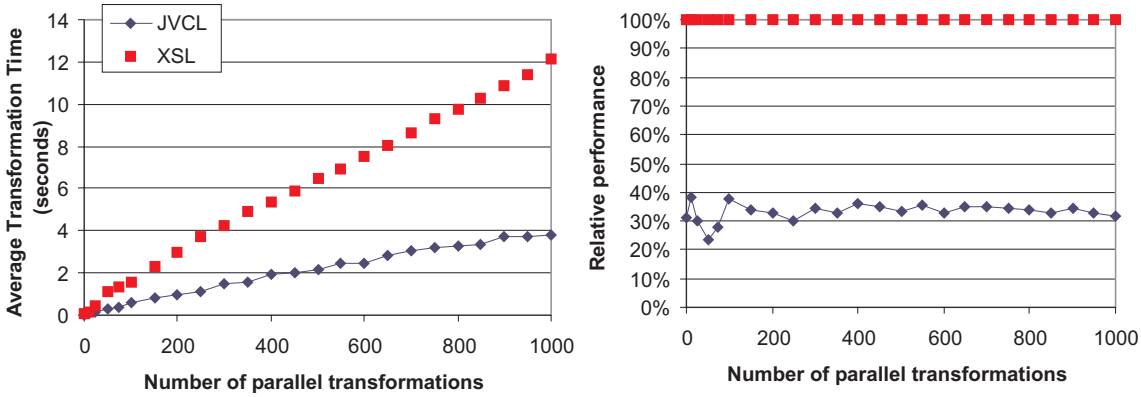


Figure 8.2.: *Performance of a visual mapping*
Absolute (left) and relative (right) performance of JOpera's visual language compared with an equivalent XSL transformation, for a variable number of transformations running in parallel.

8.2. Visual Adaptation of Mismatching Interfaces

As we have shown in the previous section, when measuring the performance of composite Web services, the results are bound by the time it takes to exchange SOAP messages across the Web. However, in case the interfaces of these services do not match, it is important to check that the transformations applied to these messages by the process engine do not significantly increase this overhead.

In this section we briefly present the results of a small experiment, whose goal was to compare the performance of the example mapping discussed in Example 4.3 on page 69 implemented using JOpera's visual language (Figure 4.11) with the performance of the equivalent XSL transformation (Figure 4.12) ran by the default XSL processor bundled with Java 1.4.1 Standard Edition. In both cases the Java virtual machine was running on a Linux v2.4.20 server with 4 XEON 1.5Ghz CPUs with hyperthreading enabled and 3.5GB of RAM. We chose this type of hardware environment in order to exploit JOpera's multithreaded execution capabilities, as it was configured to use a pool of up to 64 parallel task execution threads.

In order to minimize the effect of external factors, such as the time required to start a Java Virtual Machine, or the time necessary to load external files, all data was kept in main memory and the XSL processor was embedded inside JOpera, as one of its execution subsystems, so that the overhead of invoking it would be kept to a minimum and JOpera's multithreaded execution would not represent an unfair advantage. All transformations were applied to the same input dataset (Figure 4.11 (top)) and produced the same output dataset (Figure 4.11 (bottom))

8.2.1. Results

The results in Figure 8.2 (left) show the average transformation time as a function of the number of parallel transformations run concurrently by JOpera (going from

1 to 1000). For the transformation written in XSL (Figure 4.12) the average time grows from 50 milliseconds up to 12.1 seconds per transformation. The equivalent transformation specified visually with the process of Figure 4.11 and compiled by JOpera into Java scales better, since its average execution time grows from 15 milliseconds up to 3.8 seconds per transformation. As it can be seen from Figure 8.2 (right), the visual mapping outperforms the XSL transformation by about 35%.

8.2.2. Discussion

Our expectation was that the visual mapping, programmed in JOpera using a process with four different tasks, each of which needs to be scheduled, dispatched and executed by the system, would incur in a much higher overhead when compared with the XSL transformation, which has also been implemented within JOpera, but its process has only one task which directly calls the XSL processor as depicted in Figure 4.12 (right). Nevertheless, splitting up the transformation in three execution stages can better exploit JOpera's multithreaded execution capabilities, which come into play as the execution of each mapping is pipelined through the system. The fact that the visual mapping is compiled into Java also helps to beat the performance of the XSL processor which, instead, interprets the transformation read from the style sheet.

8.3. Scalability and Reliability

The goal of the experiments presented in this section is to analyze the performance of a significant subset of the deployment and configuration options described in Section 7.3.3 on page 160. First of all we attempt to point out the scalability limits of a centralized system, where all the data storage, event and job scheduling services are implemented using the local main memory. Then we add external persistent storage for the state information to determine what is the cost of adding persistence to the system. Then we replicate the dispatcher and navigator components, and observe the changes to the system's throughput.

Hardware Setup

The hardware and software setup for the experiments is as follows: the navigator and dispatcher kernel components were running on a cluster of dual Pentium-III 1000Mhz PCs with 1024 MB of RAM using Java 1.4.1 running on Linux v2.4.17. The three tuple space servers dedicated to state information storage, task execution scheduling and event communication were running each on separate dual Athlon 1.5Ghz with 1024 MB of RAM, Java 1.4.1 on Linux v2.4.19 and used the IBM's TSpaces implementation version 2.1.2 [108].

Variable	Values
Number of concurrent processes	1, 64, 128, 256, 512, 1024, 2048
Number of tasks	1, 10, 100
Task duration (seconds)	0, 1, 10, 30
Control flow topology	Sequential, Parallel, Matrix

Table 8.2.: *Workload Control Variables*

Workload description

The behavior of the system is affected by the properties of the workload, which are defined by the control variables listed in Table 8.2. The number of processes indicates how large is the batch of concurrent processes to be executed. The size of a process is the number of tasks composing it. We used three different process sizes: 1, 10 and 100 tasks. Larger processes require more storage space and generate a higher number of jobs and events. The duration of the tasks affects the navigator's throughput, since the longer a task runs, the longer the delay between a job startup request and the corresponding termination event. During this time the navigator(s) may be free to process other events or have to remain idle.

Finally, different topologies of the control flow of the processes generate different patterns of event exchanges. In the case of a process composed of a single task there are no degrees of freedom concerning the control flow, but as soon as the size of the process increases it is possible to connect the tasks in different ways. We have been testing our system with a variety of control flow graphs. In the case of ten tasks we used two topologies, one sequential, where the tasks are executed sequentially and a parallel one, where all tasks are executed concurrently. The same parallel topology has also been used with the larger process, composed of 100 tasks. In the case of a large process, we also tested a more complex control flow graph modeling a matrix-like computation.

Measured variables

First of all, we are interested in measuring the user perceived effect of the different configurations. This effect is measured by how long a process takes to complete. More precisely, we computed the average wall-clock time over all the concurrent processes of a certain batch.

Second, for every experiment, we recorded the batch execution time, this is how long it took to run an entire batch of concurrent processes. In the case of tasks running for 0 seconds, the execution time of the batch of processes can be used to compute the average throughput of the system, defined as the number of processed tasks per second. This value indicates the overall speed of the system in performing the operations (navigation, scheduling, running and results gathering) required to execute the tasks.

Third, in order to observe the system's internal behavior we instrumented the state information storage services to measure the time necessary to create the image

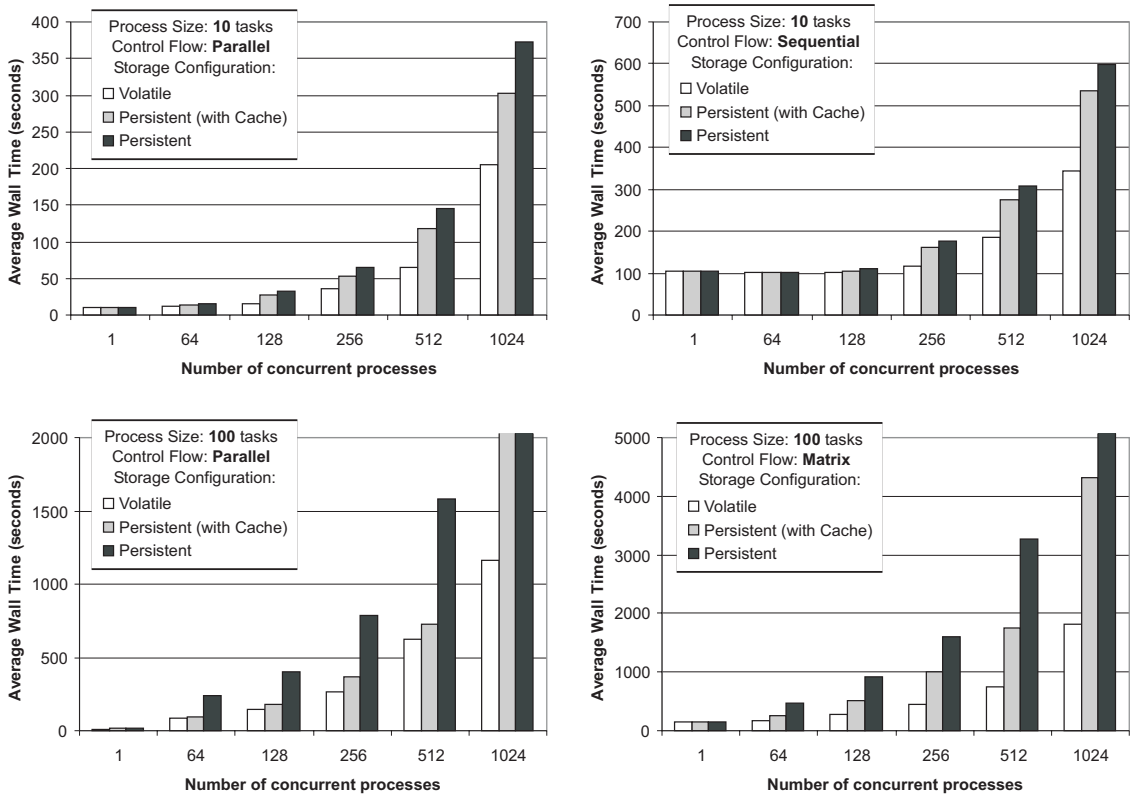


Figure 8.3.: *Performance degradation of a centralized process support system under increasingly large workloads*

of a new process instance. In our experience, this critical step is a potential performance bottleneck, since it is not possible to perform navigation until an instance has been created. We expected process instantiation to be expensive since, depending on the size of the process, (a lot of) information about the process, its tasks and their parameters needs to be written out to the state information storage service.

8.3.1. Results

Reliability and response time of a monolithic kernel

The limitations of centralized architectures can be illustrated by analyzing the performance of a centralized process support system. Such a system is built with a single component dedicated to process navigation, which uses a centralized repository to keep track of the state of the execution of the processes. As it has been often observed [124, 208], both centralization and persistence generate a significant overhead in process support systems under heavy workload. In Figure 8.3 we quantify the user perceived behavior of a centralized system while running four different types of processes.

As the results show, the system's response time, i.e., the average wall-clock dura-

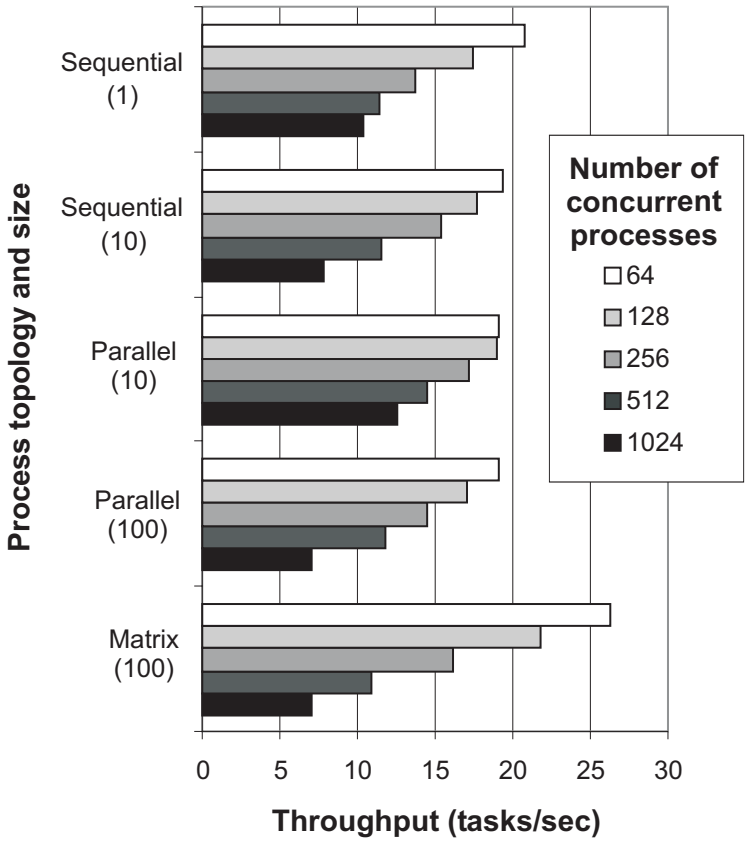


Figure 8.4.: *Throughput degradation of a centralized process support system under increasingly large workloads*

tion of a process, grows as a function of the system's workload defined as the number of processes running concurrently within the system. Relative to an unloaded system, where only one process at a time is executed, in the worst case the response time grows about 200 times when the workload size is increased thousand-fold. The actual performance degradation depends both on the type and size of the processes and on the specific properties of the system's configuration (Figure 7.4). First of all, it can be observed that a relative performance improvement can be obtained by sacrificing the reliability of the system. In fact, using the local, volatile, memory of the process navigation component to store the processes' state information, can lead to response times up to 50% shorter than the time required to perform navigation over persistent state. As discussed in Section 34 on page 155, different types of persistent storage can be used, such as a traditional relational database or, in our case, a single Tuple Space server [138].

As an attempt to combine the benefits of both configurations, we added a write-through cache located between the navigation component and the persistent storage. As the results indicate, a cache significantly reduces the penalty of using a remote storage service but still has limited scalability.

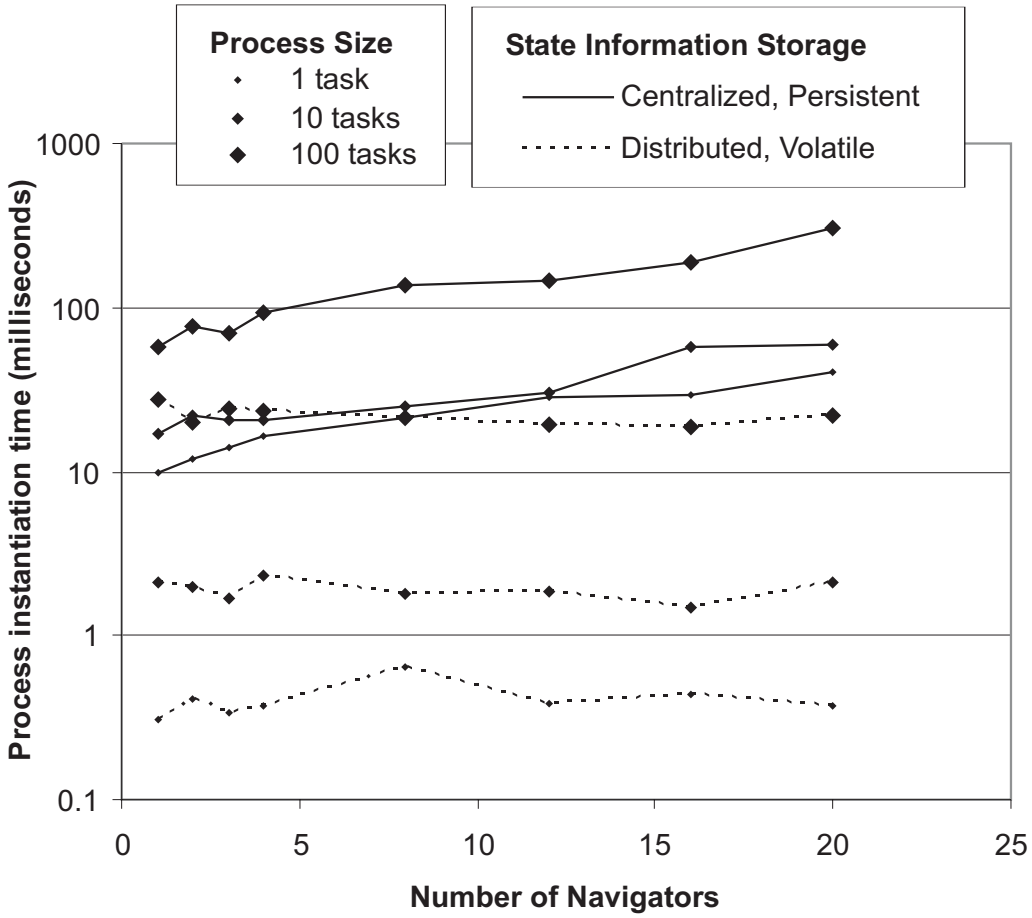


Figure 8.5.: *Scalability of the process instantiation*

Throughput degradation of a monolithic kernel

In addition to the results already presented in Figure 8.3 concerning the degradation of the response time of a centralized system under increasingly large workloads, we would like to display the corresponding throughput's degradation in Figure 8.4. This set of measurements has been performed with a monolithic kernel configured to use volatile storage and up to 64 threads for local task execution, i.e., its execution capacity is limited to 64 concurrent tasks.

For all process types, the maximum throughput is achieved when running the smallest workload. As the number of concurrent processes increases, the throughput decreases to a minimum. The actual degradation rate depends on the process topology, as the overhead of navigation is more important for larger and more complex processes.

Scalable Process Instantiation: time

Figure 8.5 displays the average process instantiation time as a function of the number of navigators, the size of the process and the configuration of the state information

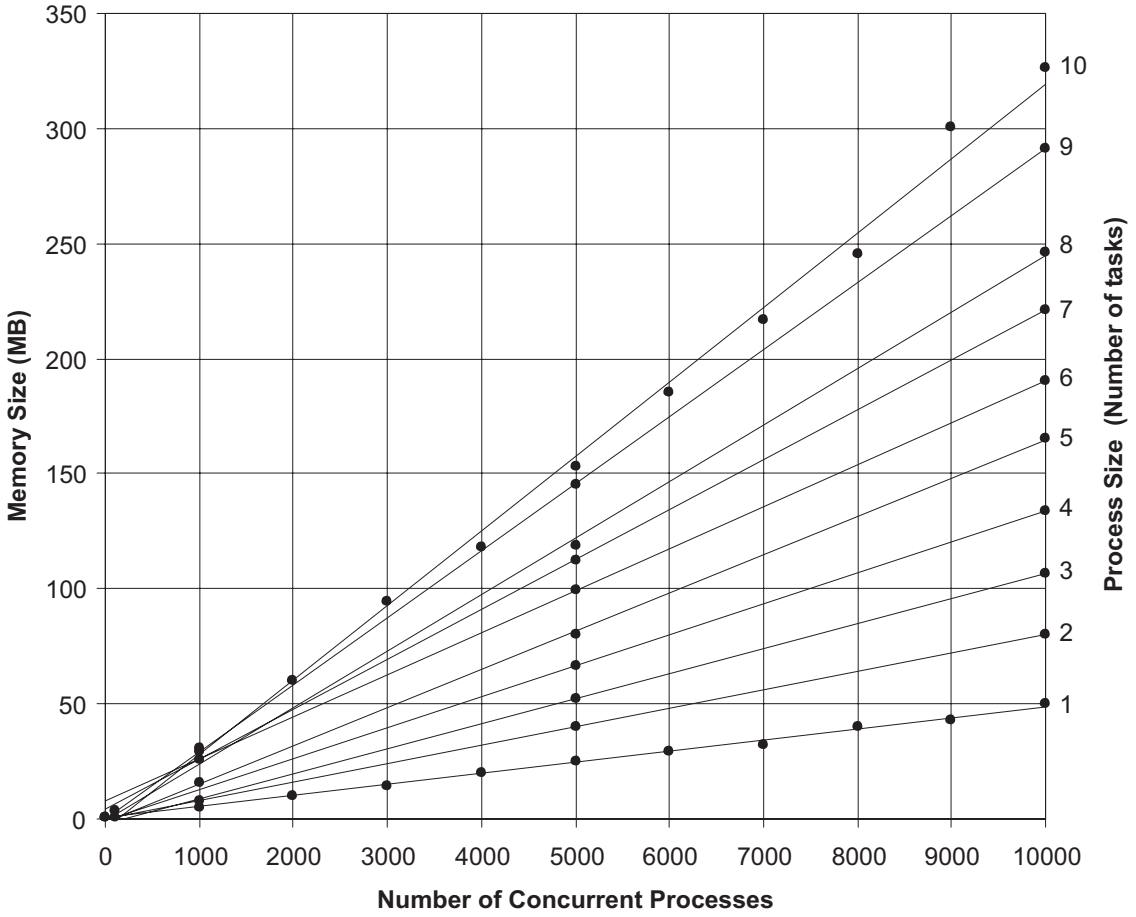


Figure 8.6.: *Memory requirements for process instantiation*

storage. As we expected, the instantiation time grows linearly with the process size: the higher number of tasks, the more information about them needs to be written to the data repository. The Figure also contains two interesting results. Not only is the instantiation time using persistent storage more than one order of magnitude longer than the time with volatile storage, but also, the volatile storage scales well with the number of navigators, since the process instantiation time remains constant. On the other hand, the performance of the centralized repository degrades as more and more navigators store in it data about their new processes. As it has been often suggested [124, 208], replicating the persistent storage would alleviate this problem. In all cases the instantiation time remains well below the 1 second boundary.

Scalable Process Instantiation: space

Complementary to the performance in the time dimension, another important aspect of the scalability of the process instantiation concerns the memory requirements. In Figure 8.6 we show the results of another set of experiments, which measure how much physical memory is required to run an increasingly large amount of process

instances. This parameter depends linearly both on the number of process instances (shown on the X-axis) and on the size of such processes, both in terms of their number of tasks (Z-axis) and parameters. In these measurements, the JOpera kernel was configured to use volatile storage, as this is the most critical configuration as far as the memory usage is concerned. As a consequence, these results should give an upper bound for other configurations, where the physical memory is only used for caching purposes.

As it can be read from Figure 8.6, in order to run 10000 processes composed of 1 task (each of which has one input and one output parameter), the JOpera kernel consumes about 50MB of RAM. A ten-fold increase in the size of the processes corresponds to an increased memory usage of more than 300MB. Clearly, these figures are not only dependent on the number of tasks per process and on the total number of parameters per process, but also on the content of these parameters, which is also stored as part of the state of a process instance. Therefore, it becomes more complex to build a model in order to predict the memory requirements of the system, given a target workload.

In case memory consumption becomes a limiting factor in the scalability of the system under a very large workload, it is possible to leverage the flexibility of the architecture to:

1. Extend the storage subsystem implementation so that the state of inactive (and completed) processes is no longer kept in main memory, but it is swapped out to secondary storage.

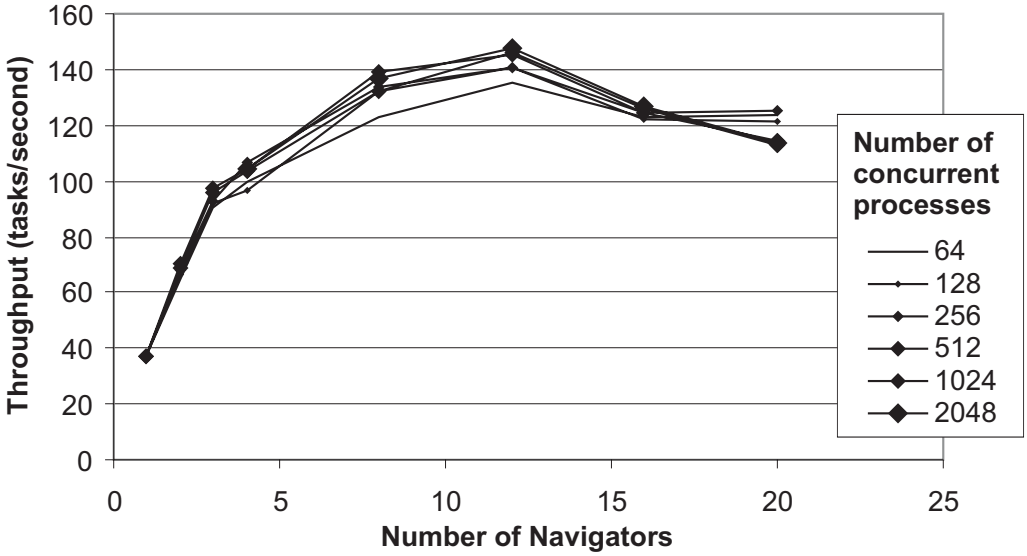
2. Replicate the kernel across a cluster of machines, so that the workload can be partitioned among each replica. This way, the total amount of memory available to JOpera can be increased beyond the capabilities of each node.

Scalable Process Navigation: throughput

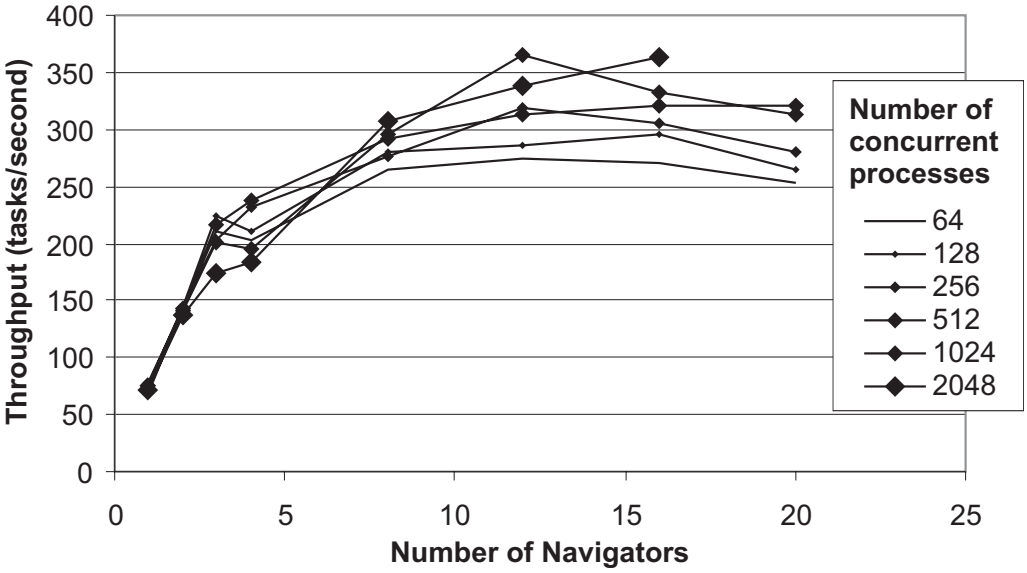
Figure 8.7 shows the average system throughput with processes of 10 parallel tasks run with a variable number of navigators, 25 dispatcher components and different workload sizes.

(a) In the case of persistent storage, for all workload sizes the throughput peaks at 12 navigators at about 140 tasks/second. This is a significant improvement with respect to a centralized system, especially considering that the throughput does not degrade as more and more processes run concurrently.

(b) The throughput actually improves as the workload size increases, indicating that, in the case of volatile storage, the performance of the replicated navigator does not saturate. Although the absolute throughput reaches about 350 tasks/second, this value is also obtained with 12 navigators, as the centralized task execution scheduler is the limiting factor of this configuration.



(a) Processes of 10 parallel tasks with persistent storage



(b) Processes of 10 parallel tasks with volatile storage

Figure 8.7.: *Scalable navigation: throughput*
Average throughput of the system using an increasingly large number of parallel navigators.

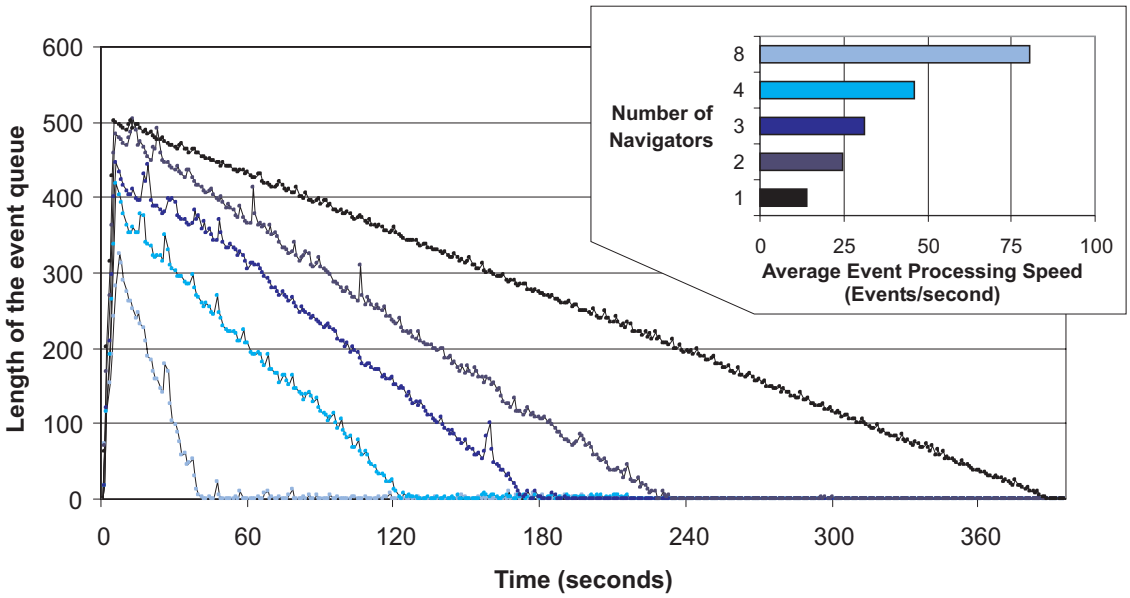


Figure 8.8.: *Scalable navigation: event queue*
Behavior of the event queue with a fixed workload of 512 processes of 10 parallel tasks of 10 seconds and a variable number of parallel navigators.

Scalable Process Navigation: event queue

In addition to a study of the overall throughput of a distributed kernel, defined as the number of tasks per second, we are also interested in observing the behavior of the event queue component, which has the critical responsibility of delivering notifications to the different navigator components.

In this experiment, the workload is kept constant at 512 processes of 10 parallel tasks, with a duration of 10 seconds each. The length of the event queue has been sampled at regular time intervals to determine the average event processing speed of the system, when the number of navigators – the consumer of events – is increased. As it is shown in Figure 8.8, the rate at which events are processed by the kernel depends linearly on the number of navigators, going from 15 events/second with one navigator up to 80 events/second with a kernel configured to use 8 navigator threads, each running on a different node of the cluster.

Looking at Figure 8.8 also suggests that the length of the event queue is a good indicator of the current workload of the system. If many events are queued, it may be possible to start additional navigator threads to process them. If the queue is empty, some navigators may be idle and can be temporarily relinquished, until the next burst of activity arrives.

Scalable Process Navigation: response time

Figure 8.9 shows the system response time with up to 2048 concurrent processes of 10 sequential tasks run in the same settings as Figure 8.7. It can be observed that

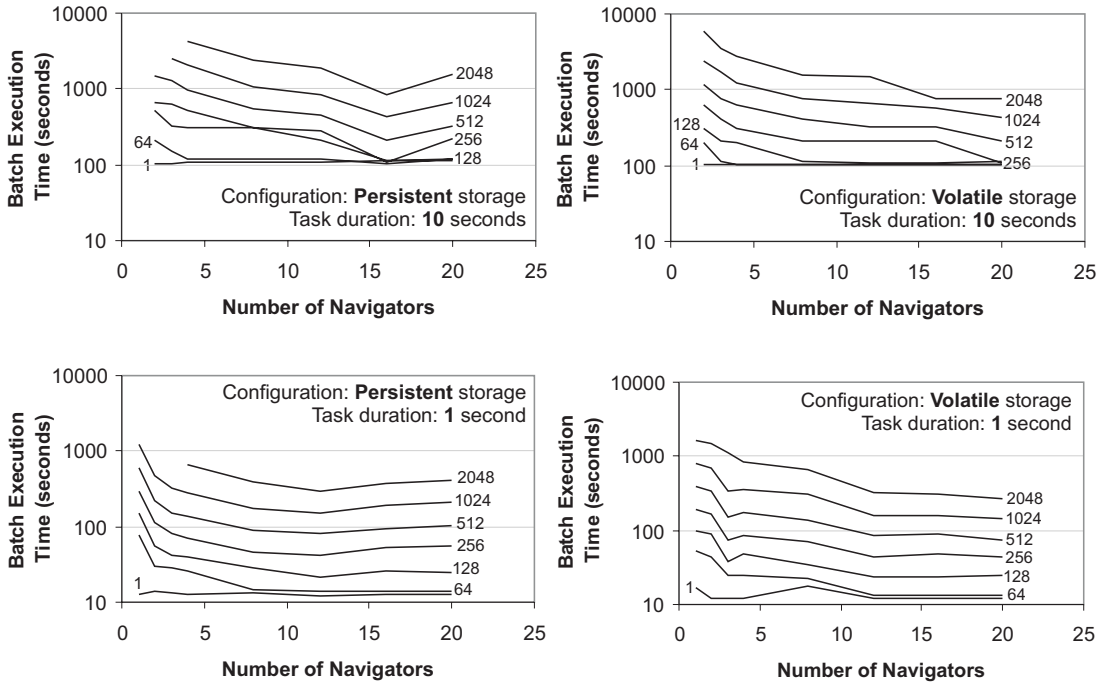


Figure 8.9.: *Scalable navigation: batch execution time*
Execution time of batches processes having 10 tasks and sequential control flow topology as a function of 4 variables: 1) the number of navigators (X-Axis), the workload size (Z-Axis), the duration of the tasks and the system configuration (volatile or persistent storage)

for small workloads, as the number of navigators increases the batch execution time approximates the time necessary to run only 1 process, which is close to 10 or 100 seconds, depending on the duration of the tasks. For larger workloads, the response time still grows linearly with the workload size, although the rate of increase can be controlled by changing the number of navigators. Using volatile storage the system scales well up to 20 navigators. Although the absolute response time is twice as high, the penalty of adding persistent storage is acceptable, as it shows good scalability up to 16 navigators accessing the same centralized data repository.

8.4. Discussion

Three orders of magnitude, this is the difference we observed between a local call to a Java method and the remote call to a Web service across the Internet. This figure gives a good indication of the potential overhead of an inflexible system, where all services that are composed are of the same type. Clearly, in many cases, it may not be possible to avoid using Web services. Especially if a process includes remote services provided by external organizations, these would not be readily accessible without this very useful technology. In this context, standards-based interoper-

ability is a necessity to enable the successful integration of external services into a process, and the price to pay for it is a fair one. However, if a system is not flexible enough to compose other types of components which are less expensive to invoke, this invocation overhead would also have to be included in the invocation of local services, for which alternative protocols may be available. With this result, described in Section 8.1 we have shown that flexibility may actually help efficiency, as it allows the system designer to choose the most appropriate mechanism to invoke a service.

In Section 8.2 we have shown a comparison of the performance of a data transformation defined visually with the equivalent XSLT transformation. Thanks to process compilation and the multi-threaded architecture of JOpera's kernel, the visual mapping outperforms by 35% the interpreted style sheet transformation. Thus, we have shown an example where a visual language is not only easier to program, but also gives better performance than an alternative, XML-based, language. Still, both solutions have been compared within the context of the JOpera service composition model, where also style sheet transformations are considered an useful component type, which can be always embedded within a process by developers which are familiar with this technology.

Finally, in Section 8.3 we have presented the performance of different deployment settings and configuration options of the JOpera kernel. More specifically, the overhead of adding reliability features has been quantified and the benefit of a flexible architecture has been shown, as the same architecture can be deployed across a cluster of computers, in order to handle large workloads.

9. Conclusion

9.1. Summary

In this dissertation we have described a visual language for service composition and an open component meta-model, followed by the design of a flexible system architecture. We have included several examples, to show how to use them to easily build and efficiently run distributed systems made out of reusable services.

The idea of following a high level approach to design and build complex systems is not new. In fact, service oriented architectures are only the latest state in the evolution of ideas related to component based software engineering, mega programming, hierarchical decomposition, middleware, coordination/glue languages affecting application domains that range from process-based e-business automation and enterprise application integration to cluster and grid computing.

With the emergence of Web services and related technologies, standards based interoperability has made it feasible to build systems out of reusable components which are potentially distributed across the Web. These loosely coupled components are provided by independent organizations in the form of *services*, i.e., their availability is guaranteed with a certain degree of transparency, where the underlying platform and languages used to implement a service are independent of the standard mechanism used to describe and access it.

Thus, service composition becomes an important aspect, where value-added services and integrated systems can be built out of aggregations of basic services in a bottom-up fashion.

Current state-of-the-art approaches to Web service composition feature process-based modeling of service interactions, ad-hoc languages, data model-driven approaches and, clearly, the addition of libraries and toolkits to traditional programming languages. Although in the first part of this dissertation we developed a language for service composition along the time dimension, which is also based on the notion of process, our approach differs from existing ones in at least the following main aspects:

1. We have defined a true *visual* service composition language (the JOpera Visual Composition Language, Chapter 3), whereas the great majority of process based languages that have been brought forward share a complementary approach based on the XML syntax. Especially from a usability perspective, this is an important difference. With JOpera developers may compose services by

drawing their interaction in a data flow graph, while the development environment keeps the underlying XML-based process model (Chapter 5) hidden at all times. However, it should be noted that a visual language does not necessarily guarantee an improved usability of the system. Issues such as visual scalability have to be taken into account. In Section 7.2.2 we have presented our approach based on a combination of nesting constructs, automatic, incremental graph layout techniques and the possibility of editing multiple, overlapping views of the same data flow graph, which support the user while browsing and editing the graphical notation.

All in all, we have argued that service composition is a good application domain for visual languages, where a simple, graph-based syntax is most appropriate to describe the order of execution of services, the data exchanges between them and the necessary failure handling behavior. Throughout the dissertation we have included several examples showing that, when compared to existing XML-based approaches, thanks to our visual language, services can be composed at a higher level of abstraction with a more intuitive syntax. Considering a scenario where services are composed automatically using additional semantics, a *surface* representation based on a visual language is still an important mean to significantly ease the understanding of the generated solution.

2. Current process modeling languages applied to Web service composition are limited to describing the composition of only one type of components: Web services. We see this as big limitation, as it greatly restricts the applicability of the composition language which – as we propose in this dissertation – should be orthogonal with respect to the mechanisms and protocols used to access the implementations of the services[1].

   Instead, we generalized the notion of service, by developing a component meta-model, in which we abstract common features shared among Web services, and many other *kinds* of services. In Chapter 4 we have attempted to show the applicability of the meta model to several different examples. With it, in a process it is possible to mix and match local and remote services which are accessible through many different mechanisms and protocols, including synchronous and asynchronous ones. Thus, freedom of choice is given to pick among services having a wide spectrum of different interoperability, performance, reliability, security, and convenience features.

   Furthermore, we have shown that generalizing the notion of service helps to keep the composition language simple. More precisely, thanks to JOpera's open component meta-model, our service composition language – describing the interactions between services at the level of their interfaces – is completely

[1]Incidentally, this does not contradict the vision behind Web services, where standards based interfaces and access protocols hide the actual implementation details. Likewise, our visual language defines composition at the level of service interfaces and does not make any assumptions about the actual access mechanisms behind them.

independent of the types of services that are composed. This way, many constructs (e.g. distinguishing synchronous from asynchronous interaction) can be shifted from the composition language to the component meta-model.

Support for heterogeneous services is not only included at the level of the composition language, but is also part of JOpera's flexible architecture, where the most efficient mechanism to access the implementation of each type of services can be selected (Section 7.4).

3. Service composition is described at the level of service interfaces by drawing a data flow graph. Most existing visual process modeling approaches overlook this important aspect, while focusing on providing a rich set of control flow patterns. Although we also included an explicit representation of the control flow of the process, we choose to emphasize data flow aspects for several reasons (Section 5.3.4), which can be summarized as follows:

   - Including an explicit, declarative model of the interactions between service interfaces helps to keep the process model at a higher level of abstraction, where the scheduling of the corresponding data flow transfers is done automatically by the compiler. Moreover, a side effect free representation (such as a data flow graph) greatly improves the clarity of the process, as it is not necessary to resort to potentially harmful global variables, like in other process modeling languages.
   - Data flow bindings between parameters of different service interfaces imply a control flow dependency between the two service invocations. From this rule, it becomes possible to derive automatically the control flow graph of a process from its data flow graph. Nevertheless, the control flow graph is also an explicit part of the process model that is used to specify additional constraints such as conditional branches, synchronization points and exception handling behavior.
   - When parameters of mismatching service interfaces are connected, type checking can be applied to guide the developer in supplying the necessary adapters. The data transformations involved can also be programmed visually using the same data flow based syntax used to compose the services.

After giving a detailed description with several examples of the JOpera Visual Composition Language (Chapter 3), JOpera's component meta-model (Chapter 4) and the underlying Opera process Modeling Language (Chapter 5), in the second part of the dissertation we described the JOpera system which provides an integrated set of tools to support the rapid development of composite services using the aforementioned languages.

It is our view that visual, process based service composition tools will not gain a widespread acceptance if they cannot deliver a level of performance comparable

to traditional programming languages. In other words, the benefit of using such languages and tools with their ability to model service composition at a higher level of abstraction should not be reduced (or lost) at runtime, when the models need to be executed. To achieve this objective, in the design of the architecture of the JOpera system we faced two competing requirements: efficiency and flexibility.

To achieve a higher efficiency during the execution of the processes, we built on the idea of using *compiled* process execution. As opposed to the *interpreted* process execution approach followed by most process engines (or process interpreters), in JOpera, process descriptions are compiled to Java executable code, which is then dynamically loaded into the process execution kernel to manage the execution of multiple, concurrent process instances. As we discussed in Chapter 6 the generated code is still flexible enough to support dynamic late binding of service interfaces to their implementations.

Flexibility is an important requirement when building a system to execute processes composed out of services, whose type is not known in advance. Thus, in this case, flexibility does not contradict efficiency. Flexibility enables to choose the most efficient access mechanism to the service implementation among several alternative ones. Instead, a rigid architecture, limited to invoking services of only one component type, would force to use the same mechanism with all service invocations, preventing the possibility of doing optimizations. As we have shown in Section 8.1, the overhead in invoking remote Web services is two to three orders of magnitude larger than the one involved in a Java method call within the same Java virtual machine. Therefore, depending on the location and the granularity of the services to be composed, the flexibility of choosing the most appropriate access mechanism, has a potentially beneficial impact on the performance of the final composite system.

Furthermore, we have shown that JOpera is a flexible system as it can be deployed in several different settings and configurations (Section 7.3.3). This way, the most useful solution to the trade-off between costs and benefits of several features (e.g. quality of service guarantees, reliability, scalability) can be found at deployment time, depending on external requirements and environmental settings. For example, in some usage scenarios (e.g., at development time), persistence is not necessary. Thus, the JOpera kernel can be deployed in a volatile configuration, which offers a smaller runtime overhead. Likewise, the flexible architecture of JOpera supports dynamic reconfiguration. This feature represents an important step towards building a self-tuning, autonomous system, where the optimal configuration is determined automatically [21].

As a concluding remark, with JOpera it is possible to rapidly develop and run complex, distributed applications made out of many different types of components (including, but not limited to Web services). By using a visual glue language, such applications can be built by literally drawing the interactions between the interfaces of the services composing them. Thus, loosely coupled services which have been originally developed independently can be adapted and integrated into more complex, value-added services. It remains an interesting study to observe the effect of the composition on its components, which are no longer free to evolve

independently. On the one hand, the reliability of the composite system is limited by the weakest of its component services (Example 3.4). On the other hand, it becomes difficult to modify the interfaces of the services without affecting the ones depending on them (Example 4.3).

9.2. Outlook

The ideas about the languages and the system presented in this dissertation can be extended in several research directions. It would be also interesting to apply them to other domains, different from the examples that we have included throughout the dissertation.

- First of all, additional language features could be included in the JOpera Visual Composition Language (Chapter 3). These concern its very simple control flow syntax, which – if necessary – could be extended with constructs for explicitly modeling branches and merges in the control flow graph. However, these would not add to the expressiveness of the language, but only provide so-called "syntactical sugar". With them, some of the features already part of the underlying process model would be also have a visual representation. Likewise, the *swimlanes* construct should be evaluated in order to compare it with the current approach based on system parameters. Swimlanes could be used to visually assign the owner of a task (or the provider of a service) by positioning the task inside the swimlane corresponding to the owner. Although with a different syntax, it is already possible to specify this information by using system parameters. As part of this evaluation, the visual scalability of such swimlanes should be taken into account. It is not clear, after all, whether this construct could be applied with success to scenarios involving the interaction of a large number of parties. In addition to sequential and parallel list-based loops, an intermediate approach based on pipelines could also be an interesting extension. Similar to [25], the values of the input list would be processed by the tasks included in the loop's body in a pipelined fashion.

- Assuming the availability of appropriate standards to specify the semantics associated to an interface. Moreover, assuming that the corresponding tools to use this information for producing goal-directed, automatic service composition exist. Then, the JOpera visual development environment could be extended by integrating such functionality. For instance, it would become possible – up to a certain extent – to provide features such as automatic process completion or automatic generation of mappings between mismatching service interfaces.

- With the aim of increasing the heterogeneity of the services that can be composed in a process, the list of component types supported by JOpera presented in Chapter 4 should be extended further. This way, the validity of JOpera's component meta-model would be confirmed and the system could be applied

to a even wider set of domains. For example, by adding support for Grid services (as opposed to Web services), processes could be used to build Grid-based distributed computations.

- In Chapter 5 the syntax of the Opera process Modeling Language has been defined as a particular kind of XML syntax. However, the semantics of the Opera process Modeling Language was defined operationally, as embedded into the corresponding compilers (Chapter 6). Thus, producing a formal description of this semantics, based, e.g., on the π-calculus [162] would be a useful line of work, as many existing model checking tools based on such formal representations could augment the current set of available tools.

- The topic of modeling service *conversations* was only briefly touched by this dissertation. On the one hand, a process can be used in a straightforward manner for modeling a particular interaction between two or more parties based on synchronous and asynchronous message exchanges (Section 4.10). On the other hand, due to the lack of a standardized way of modeling such interaction protocols, it is still difficult to provide automatic tools that can statically verify that a process represents a valid instance of a given conversation.

- We claim that the architecture of JOpera's kernel presented in Chapter 7 is flexible enough that several different implementations of its internal system components can be used without modifying the structure of the system, and, most important, without affecting the compiler. Therefore, within the framework of such flexible architecture, it is possible to implement critical system components – the ones implementing abstractions such as the task scheduler, the event queues and the state information storage – using different technologies (e.g., peer to peer systems, replicated databases, messaging systems). With this, it becomes possible to perform a detailed analysis of the level of performance that can be achieved with each solution. These results can be compared with the deployment constraints and additional features provided by each implementation technology in order to determine what are the requirements for providing given quality of service guarantees.

A. Opera Modeling Language Schema

Here is the formal definition of the Opera Modeling Language first described in Chapter 5 written using the XML Schema notation [247].

```
<?xml version="1.0" encoding="UTF-8"?>

<!--

XML Schema for the Opera Modeling Language (OML)
version 2.0
(C)2001-2004 Cesare Pautasso, ETH Zurich
Fri Apr 16 15:06:03 2004

-->

<xs:schema targetNamespace="http://www.iks.ethz.ch/jopera/oml"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">


<!-- Root Element -->

<xs:element name="OCR">
 <xs:annotation><xs:documentation>
 Root Element: The root element of an Opera Markup Language document
 </xs:documentation></xs:annotation>
 <xs:complexType>
    <xs:attribute name="VER" type="xs:string" use="required"/><!-- Version -->
    <xs:attribute name="MAXID" type="xs:integer" use="required"/><!-- NextObjectID -->
    <xs:all>
     <xs:element name="PROCS" minOccurs="0" maxOccurs="1"><!-- Processes -->
      <xs:sequence>
       <xs:element ref="PROC" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
     <xs:element name="PROGRAMS" minOccurs="0" maxOccurs="1"><!-- Programs -->
      <xs:sequence>
       <xs:element ref="PROGRAM" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
     <xs:element name="COMPS" minOccurs="0" maxOccurs="1"><!-- ComponentTypes -->
      <xs:sequence>
       <xs:element ref="COMP" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
    </xs:all>
 </xs:complexType>
</xs:element>

<!-- Abstract Elements -->
```

```
<xs:element name="Object">
 <xs:annotation><xs:documentation>
 Object Element: Any object with an ID. If not specified, all other elements extend this basic one, as every element in an OML document must be uniquely identifiable
 </xs:documentation></xs:annotation>
 <xs:complexType>
    <xs:attribute name="OID" type="xs:ID" use="required"/><!-- ObjectID -->
    <xs:attribute name="CS" type="CompilerState" use="required"/><!-- CompilerState -->
 </xs:complexType>
</xs:element>

<xs:simpleType name="CompilerState">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Ok"/>
  <xs:enumeration value="Comment"/>
  <xs:enumeration value="Error"/>
  <xs:enumeration value="Warning"/>
 </xs:restriction>
</xs:simpleType>

<xs:element name="NamedObject">
 <xs:annotation><xs:documentation>
 NamedObject Element: It represents any element with a name and an optional description
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="Object">
    <xs:attribute name="NAME" type="xs:string" use="required"/><!-- Name -->
    <xs:attribute name="DESC" type="xs:string"/><!-- Description -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="Interface">
 <xs:annotation><xs:documentation>
 Interface Element: This element represents the interface of any process element which can exchange data. The interface of a Process or a Program consists of input and output lists of user-defined parameters. In case of ComponentTypes and AccessMethods, such parameter lists are used for defining system parameters
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="NamedObject">
    <xs:attribute name="AUTHOR" type="xs:string" use="required"/><!-- Author -->
    <xs:all>
     <xs:element name="INBOX" minOccurs="0" maxOccurs="1"><!-- Inbox -->
      <xs:sequence>
       <xs:element ref="PARAM" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
     <xs:element name="OUTBOX" minOccurs="0" maxOccurs="1"><!-- Outbox -->
      <xs:sequence>
       <xs:element ref="PARAM" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
    </xs:all>
  </xs:extension>
 </xs:complexType>
</xs:element>

<!-- Process Elements -->

<xs:element name="PROC">
 <xs:annotation><xs:documentation>
 Process Element: A Process contains a number of tasks connected by data and control flow graphs, which are defined visually. In JOpera, a process is the smallest executable unit of composition. It describes how a set of service invocations are composed together. Through the SubProcess construct, processes can also be directly reused as components within other processes. Similarly, processes can be published as Web services for external reuse. To model and control its level of reuse, a process has the following attributes
 </xs:documentation></xs:annotation>
 <xs:complexType>
```

```
  <xs:extension base="Interface">
    <xs:attribute name="PUBLISHED" type="xs:boolean" use="required"/><!-- Published -->
    <xs:attribute name="SUBPROC" type="xs:boolean" use="required"/><!-- SubProcess -->
    <xs:attribute name="ABSTRACT" type="xs:boolean" use="required"/><!-- Abstract -->
    <xs:all>
     <xs:element name="VIEWS" minOccurs="0" maxOccurs="1"><!-- Views -->
      <xs:sequence>
       <xs:element ref="VIEW" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
     <xs:element name="DATAFLOW" minOccurs="0" maxOccurs="1"><!-- Dataflow -->
      <xs:sequence>
       <xs:element ref="BIND" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
     <xs:element name="CONSTS" minOccurs="0" maxOccurs="1"><!-- Constants -->
      <xs:sequence>
       <xs:element ref="PARAM" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
     <xs:element name="TASKS" minOccurs="0" maxOccurs="1"><!-- Tasks -->
      <xs:sequence>
       <xs:element ref="Task" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
    </xs:all>
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="Task">
 <xs:annotation><xs:documentation>
 Task Element: A task represent any component of a process: a basic step in the computation modeled by a process. The description of how such steps depend on each other is also part of the content of a task. More precisely, its Activator and Condition attributes model the basic control flow dependencies linking a task to its predecessors. The remaining attributes are used for describing more advanced scheduling and synchronization options, in case the task belongs to a list-based loop, i.e., it is found within a pair of split and merge operators. The concrete forms of a task, which appear in an OML document, are either the Activity or the Sub-Process elements, which contain additional information modeling the actual service to be invoked as part of the task execution
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="NamedObject">
    <xs:attribute name="ACT" type="xs:string" use="required"/><!-- Activator -->
    <xs:attribute name="COND" type="xs:string" use="required"/><!-- Condition -->
    <xs:attribute name="PRIORITY" type="xs:integer"/><!-- Priority -->
    <xs:attribute name="DEP" type="DependencyType"/><!-- Dependency -->
    <xs:attribute name="SYNCH" type="SynchType"/><!-- Synchronization -->
    <xs:attribute name="FAILH" type="FailureHandlingType"/><!-- FailureHandling -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:simpleType name="DependencyType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Finished"/>
  <xs:enumeration value="Failed"/>
  <xs:enumeration value="FinishOrFailed"/>
  <xs:enumeration value="Aborted"/>
  <xs:enumeration value="None"/>
 </xs:restriction>
</xs:simpleType>

<xs:simpleType name="SynchType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="WaitForAll"/>
  <xs:enumeration value="WaitForOne"/>
 </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="FailureHandlingType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="FailForOne"/>
  <xs:enumeration value="FailForAll"/>
  <xs:enumeration value="FailForPercent"/>
 </xs:restriction>
</xs:simpleType>

<xs:element name="ACTIVITY">
 <xs:annotation><xs:documentation>
 Activity Element: An Activity is a task which references a program
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="Task">
    <xs:attribute name="PROGRAMID" type="xs:IDREF" use="required"/><!-- ProgramID -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="SUBPROC">
 <xs:annotation><xs:documentation>
 SubProcess Element: A SubProcess is a task which references a process.
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="Task">
    <xs:attribute name="PROCESSID" type="xs:IDREF" use="required"/><!-- ProcessID -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<!-- Data Flow Elements -->

<xs:element name="PARAM">
 <xs:annotation><xs:documentation>
 Parameter Element: A Parameter models a data container that can be attached to a Process, a Program, an AccessMethod and a ComponentType.
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="NamedObject">
    <xs:attribute name="TYPE" type="xs:string"/><!-- Type -->
    <xs:attribute name="VALUE" type="xs:string"/><!-- Value -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="BIND">
 <xs:annotation><xs:documentation>
  Binding Element: A data flow Binding is an edge linking a source and a destination parameter in the data flow graph of the Process. Parameters can belong to a process (including constants), or a task. For tasks, the parameters are defined in the referenced program/process. By default, a binding models a data transfer between a pair of parameters. Additionally, using the ActionType and ActionData attributes, it is possible to specify the application of a split or merge operator to the data in transit.
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="Object">
    <xs:attribute name="SOURCETYPE" type="BindRefType" use="required"/><!-- SourceType -->
    <xs:attribute name="SOURCEPID" type="xs:IDREF" use="required"/><!-- SourceParam -->
    <xs:attribute name="DESTTYPE" type="BindRefType" use="required"/><!-- DestType -->
    <xs:attribute name="DESTPID" type="xs:IDREF" use="required"/><!-- DestParam -->
    <xs:attribute name="ACTION" type="ActionType"/><!-- ActionType -->
    <xs:attribute name="ACTIONDATA" type="xs:string"/><!-- ActionData -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:simpleType name="BindRefType">
 <xs:restriction base="xs:string">
```

```
  <xs:enumeration value="Normal"/>
  <xs:enumeration value="System"/>
  <xs:enumeration value="Constant"/>
 </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ActionType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Copy"/>
  <xs:enumeration value="Split"/>
  <xs:enumeration value="Merge"/>
 </xs:restriction>
</xs:simpleType>

<!-- JOpera Visual Composition Language (JVCL) Elements -->

<xs:element name="VIEW">
 <xs:annotation><xs:documentation>
 View Element: A view is a visual representation of the control flow or data flow graphs of the enclosing process element.  A graph is modeled both in terms of its topology (edges linking nodes) as well as its two dimensional layout (nodes are displayed as rectangles located at certain coordinates, while edges can be routed following a set of control points)
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="NamedObject">
    <xs:attribute name="VTYPE" type="ViewType" use="required"/><!-- ViewType -->
    <xs:attribute name="PAPER" type="xs:string" use="required"/><!-- PaperSize -->
    <xs:all>
     <xs:element name="ARROWS" minOccurs="0" maxOccurs="1"><!-- Arrows -->
      <xs:sequence>
       <xs:element ref="ARROW" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
     <xs:element name="BOXES" minOccurs="0" maxOccurs="1"><!-- Boxes -->
      <xs:sequence>
       <xs:element ref="Box" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
     <xs:element name="GROUPS" minOccurs="0" maxOccurs="1"><!-- Groups -->
      <xs:sequence>
       <xs:element ref="GROUPBOX" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
    </xs:all>
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:simpleType name="ViewType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Controlflow"/>
  <xs:enumeration value="Dataflow"/>
 </xs:restriction>
</xs:simpleType>

<xs:element name="ViewObject">
 <xs:annotation><xs:documentation>
 ViewObject Element: Any element inside a View element should extend this one. In the case of Arrows and RefBoxes, it represents a visible object which references an element in the rest of the process model.
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="Object">
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="GROUPBOX">
 <xs:annotation><xs:documentation>
```

```
 GroupBox Element: A Group is a semantically transparent grouping of visual objects used to constrain the automatic layout algorithms
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="ViewObject">
    <xs:attribute name="ELEMENTS" type="xs:IDREF" use="required"/><!-- Elements -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="Box">
 <xs:annotation><xs:documentation>
 Box Element: A box is any rectangle displayed in a view, it is further specialized by the TextBox and RefBox concrete elements. All coordinates of the graphic elements are stored using floating point values. Although the origin of the coordinate system should be mapped to the center of the screen, no explicit assumption is made about the direction of the X and Y axis
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="ViewObject">
    <xs:attribute name="X" type="Float" use="required"/><!-- X -->
    <xs:attribute name="Y" type="Float" use="required"/><!-- Y -->
    <xs:attribute name="DX" type="Float" use="required"/><!-- Width -->
    <xs:attribute name="DY" type="Float" use="required"/><!-- Height -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="RBOX">
 <xs:annotation><xs:documentation>
 RefBox Element: A RefBox is a node of the graph, which represents a task, a parameter or the process itself. The text shown inside the RefBox is extracted from the referred document element. These boxes are automatically resized to fit with the displayed text
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="Box">
    <xs:attribute name="REF" type="xs:IDREF" use="required"/><!-- Reference -->
    <xs:attribute name="REFTYPE" type="BRefType" use="required"/><!-- ReferenceType -->
    <xs:all>
     <xs:element name="BOXES" minOccurs="0" maxOccurs="1"><!-- Boxes -->
      <xs:sequence>
       <xs:element ref="RBOX" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
    </xs:all>
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:simpleType name="BRefType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Activity"/><!-- Activity -->
  <xs:enumeration value="SubProcess"/><!-- SubProcess -->
  <xs:enumeration value="DataInBox"/><!-- Input Parameter -->
  <xs:enumeration value="DataOutBox"/><!-- Output Parameter -->
  <xs:enumeration value="ProcessInput"/><!-- Process Input Placeholder -->
  <xs:enumeration value="ProcessOutput"/><!-- Process Output Placeholder -->
  <xs:enumeration value="Const"/><!-- Constant Parameter -->
  <xs:enumeration value="SysInBox"/><!-- System Input Parameter -->
  <xs:enumeration value="SysOutBox"/><!-- System Output Parameter -->
 </xs:restriction>
</xs:simpleType>

<xs:element name="TEXTBOX">
 <xs:annotation><xs:documentation>
 TextBox Element: A TextBox is used to display textual comments inside a view. If a text box overlaps with any other ViewObjects, such objects appear as commented out and the corresponding process elements are ignored by the compiler
 </xs:documentation></xs:annotation>
```

```
 <xs:complexType>
  <xs:extension base="Box">
    <xs:attribute name="TEXT" type="xs:string" use="required"/><!-- Text -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="ARROW">
 <xs:annotation><xs:documentation>
 Arrow Element: An arrow is a directed edge linking two boxes. Depending on the type of the container View element, it represents a control flow dependency between two task boxes, or a data flow binding between a pair of parameters. An arrow cannot exist without both of the boxes which it links
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="ViewObject">
    <xs:attribute name="ID1" type="xs:IDREF" use="required"/><!-- Source -->
    <xs:attribute name="ID2" type="xs:IDREF" use="required"/><!-- Destination -->
    <xs:attribute name="CONTROLPOINTS" type="xs:string"/><!-- ControlPoints -->
    <xs:attribute name="REF" type="xs:IDREF"/><!-- Reference -->
    <xs:attribute name="REFTYPE" type="ARefType" use="required"/><!-- ReferenceType -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:simpleType name="ARefType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Copy"/>
  <xs:enumeration value="Split"/>
  <xs:enumeration value="Merge"/>
  <xs:enumeration value="Finished"/>
  <xs:enumeration value="Failed"/>
  <xs:enumeration value="FinishOrFail"/>
  <xs:enumeration value="Aborted"/>
  <xs:enumeration value="Unreachable"/>
 </xs:restriction>
</xs:simpleType>

<!-- Program Library Elements -->

<xs:element name="PROGRAM">
 <xs:annotation><xs:documentation>
 Program Element: A Program is a component referred by the activities of a process and represents the invocation of a service.  Similar to WSDL, a Program both includes information on the interface  of a service, as well as multiple AccessMethod elements which refer  to the actual component type to be used while accessing the service.
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="Interface">
    <xs:attribute name="SIZE" type="xs:integer" use="required"/><!-- Size -->
    <xs:attribute name="RESTART" type="xs:integer" use="required"/><!-- Restart -->
    <xs:all>
     <xs:element name="ACCESS" minOccurs="0" maxOccurs="1"><!-- Access Methods -->
      <xs:sequence>
       <xs:element ref="METHOD" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:element>
    </xs:all>
  </xs:extension>
 </xs:complexType>
</xs:element>

<xs:element name="METHOD">
 <xs:annotation><xs:documentation>
 AccessMethod Element: An access method contains the system parameters that control how to access the services provided by the specified component type. It also defines a mapping between the program input parameters -- which are application dependent and defined by the user  -- to the system input parameters of the ComponentType, which depend on the mechanisms and the protocols used to perform the service invocation
 </xs:documentation></xs:annotation>
```

```
 <xs:complexType>
  <xs:extension base="Interface">
    <xs:attribute name="COMP" type="xs:IDREF" use="required"/><!-- ComponentType -->
  </xs:extension>
 </xs:complexType>
</xs:element>

<!-- Component Type Modeling Elements -->

<xs:element name="COMP">
 <xs:annotation><xs:documentation>
 ComponentType Element: A Component Type is a template for an AccessMethod, it defines a set of input and output system parameters that are used to generate the ones of the access method referencing it
 </xs:documentation></xs:annotation>
 <xs:complexType>
  <xs:extension base="Interface">
    <xs:attribute name="PARENT" type="xs:IDREF" use="required"/><!-- Extends -->
    <xs:attribute name="ABSTRACT" type="xs:boolean" use="required"/><!-- Abstract -->
  </xs:extension>
 </xs:complexType>
</xs:element>

</xs:schema>
```

B. JOpera Compiler Output

In this appendix we include the full listing of the Java executable code produced by JOpera's compiler applied to the Example 4.1 on page 55. For more information on how compilation works, please turn to Section 6.5.4 on page 137.

```
 1
 2 //JOpera Process Template Plugin
 3 //OML2Java Compiler Version v1.65 20040130
 4
 5 package ch.ethz.bioopera.templates;
 6
 7 import ch.ethz.bioopera.kernel.*;
 8 import ch.ethz.bioopera.ws.tools.*;
 9 import ch.ethz.bioopera.common.*;
10 import java.util.*;
11
12 public class TStockQuoteConvertTemplate extends Template
13 {
15   public String getName()
16   {
17     return "StockQuoteConvert";
18   }
19
20   public String getAuthor()
21   {
22     return "CP";
23   }
24
25   public String getDescription()
26   {
27     return "This process returns the 20-minute delayed stock quote of a given stock market
symbol, converted to the currency of the given country.";
28   }
29
30   public String getCompileDate()
31   {
32     return "2004-02-06 15:45:32.937";
33   }
34
35   public void Evaluate(TID Context) throws MemoryException
36   {
37     boolean part_ok;
38     int nc;
39     State State_PROC;
40
41     TID Context_PROC = PROC(Context);
42     TID Context_TASK_getStockQuote = TASK(Context, "getStockQuote");
43     TID Context_TASK_getExchangeRate = TASK(Context, "getExchangeRate");
44     TID Context_TASK_Multiply = TASK(Context, "Multiply");
45
46     State_PROC = Memory.getState(Context_PROC);
47
48     if (State_PROC == State.INITIAL)
49     {
50       TimeStamp(Context_PROC, Box.StartTime);
```

```
52      Memory.Copy(MakeAddress(Context_PROC, Box.Input, "symbol"),
53            MakeAddress(Context_TASK_getStockQuote, Box.Input, "symbol"));
54
55      InputParams.clear();
56      InputParams.put("symbol", Memory.Load(MakeAddress(  Context_TASK_getStockQuote,
57                                Box.Input,
58                                "symbol")));
59
60      TimeStamp(Context_TASK_getStockQuote, Box.StartTime);
61      Exec.Start(Context_TASK_getStockQuote, InputParams);
62
63      Memory.Copy(  MakeAddress(Context_PROC, Box.Input, "country"),
64              MakeAddress(Context_TASK_getExchangeRate,
65                    Box.Input,
66                    "country2"));
67
68      InputParams.clear();
69      InputParams.put("country1", Memory
70          .Load(MakeAddress(  Context_TASK_getExchangeRate,
71                    Box.Input,
72                    "country1")));
73      InputParams.put("country2", Memory
74          .Load(MakeAddress(  Context_TASK_getExchangeRate,
75                    Box.Input,
76                    "country2")));
77
78      String p_country1 = (String) InputParams.get("country1");
79      String p_country2 = (String) InputParams.get("country2");
80
81      if (!(p_country1.equals(p_country2)))
82      {
83        TimeStamp(Context_TASK_getExchangeRate, Box.StartTime);
84        Exec.Start(Context_TASK_getExchangeRate, InputParams);
85      }
86      else
87      {
88        Memory.setState(Context_TASK_getExchangeRate, State.UNREACHABLE);
89      }
90      Memory.setState(Context_PROC, State.RUNNING);
91    }
92    else
93    {
94      State State_Context = Memory.getState(Context);
95      if ((State_PROC == State.RUNNING)
96      || (State_Context == State.FINISHING)
97      || (State_Context == State.FAILED)
98      || (State_Context == State.UNREACHABLE))
99      {
100       State State_getStockQuote = Memory.getState(Context_TASK_getStockQuote);
101       State State_getExchangeRate = Memory.getState(Context_TASK_getExchangeRate);
102       State State_Multiply = Memory.getState(Context_TASK_Multiply);
103
104       if ((State_getStockQuote == State.FINISHING))
105       {
106         Memory.Store(MakeAddress(  Context_TASK_getStockQuote,
107                       Box.Output,
108                       "Result"), (String) Results.get("Result"));
109         Memory.Copy(MakeAddress(Context_TASK_getStockQuote,
110                     Box.Output,
111                     "Result"),
112               MakeAddress(Context_PROC, Box.Output, "quote"));
113
114         Memory.setState(Context_TASK_getStockQuote, State.FINISHED);
115         State_getStockQuote = State.FINISHED;
116       }
117
118       if ((State_getExchangeRate == State.FINISHING))
119       {
```

```
120           Memory.Store(MakeAddress(  Context_TASK_getExchangeRate,
121                        Box.Output,
122                        "Result"), (String) Results.get("Result"));
124           Memory.setState(Context_TASK_getExchangeRate, State.FINISHED);
125           State_getExchangeRate = State.FINISHED;
126         }
127
128         if (State_Multiply == State.INITIAL)
129         {
130           if ((State_getStockQuote == State.FINISHED)
131           && (State_getExchangeRate == State.FINISHED))
132           {
133             Memory.Copy(MakeAddress(Context_TASK_getExchangeRate,
134                         Box.Output,
135                         "Result"),
136                   MakeAddress(Context_TASK_Multiply,
137                         Box.Input,
138                         "b"));
139             Memory.Copy(MakeAddress(Context_TASK_getStockQuote,
140                         Box.Output,
141                         "Result"),
142                   MakeAddress(Context_TASK_Multiply,
143                         Box.Input,
144                         "a"));
145
146             InputParams.clear();
147             InputParams.put("a", Memory
148                 .Load(MakeAddress(  Context_TASK_Multiply,
149                           Box.Input,
150                           "a")));
151             InputParams.put("b", Memory
152                 .Load(MakeAddress(  Context_TASK_Multiply,
153                           Box.Input,
154                           "b")));
155
156             TimeStamp(Context_TASK_Multiply, Box.StartTime);
157             Exec.Start(Context_TASK_Multiply, InputParams);
158           }
159           else
160           if (((State_getStockQuote != State.FINISHED)
161            && (State_getStockQuote != State.INITIAL)
162           && (State_getStockQuote != State.RUNNING)
163           && (State_getStockQuote != State.WAITING)
164           && (State_getStockQuote != State.FINISHING))
165           || ((State_getExchangeRate != State.FINISHED)
166           && (State_getExchangeRate != State.INITIAL)
167           && (State_getExchangeRate != State.RUNNING)
168           && (State_getExchangeRate != State.WAITING)
169           && (State_getExchangeRate != State.FINISHING)))
170           {
171             Memory.setState(Context_TASK_Multiply,
172                     State.UNREACHABLE);
173           }
174         }
176         if ((State_Multiply == State.FINISHING))
177         {
178           Memory.Store(MakeAddress(  Context_TASK_Multiply,
179                        Box.Output,
180                        "result"), (String) Results.get("result"));
181
182           Memory.Copy(MakeAddress(Context_TASK_Multiply,
183                       Box.Output,
184                       "result"),
185                 MakeAddress(Context_PROC, Box.Output, "quote"));
186
187           Memory.setState(Context_TASK_Multiply, State.FINISHED);
188           State_Multiply = State.FINISHED;
189         }
```

```
190
191         if (((State_Multiply == State.FINISHED) || (State_Multiply == State.UNREACHABLE)))
192         {
193           if (State_PROC != State.FINISHED)
194               Memory.setState(Context_PROC, State.FINISHED);
195         }
197         if ((State_getStockQuote == State.FAILED)
198         || (State_getExchangeRate == State.FAILED)
199         || (State_Multiply == State.FAILED))
200         {
201           if (State_PROC != State.FAILED)
202               Memory.setState(Context_PROC, State.FAILED);
203         }
204       }
205
206       if ((State_PROC == State.FINISHED) || (State_PROC == State.FAILED))
207       {
208         Results.clear();
209         Results.put("quote", Memory.Load(MakeAddress(  Context_PROC,
210                                  Box.Output,
211                                  "quote")));
213         Completed(Context_PROC);
214       }
215     }
216   }
217
218   public void SetupImage(TID Context, Map Params)
219   {
220     SetupSystemBox(PROC(Context));
221     SetupParam(PROC(Context), Box.Input, "symbol", Params.get("symbol"));
222     SetupParam(PROC(Context), Box.Input, "country", Params.get("country"));
223     SetupParam(PROC(Context), Box.Output, "quote", "");
225     TID Context_TASK_getStockQuote = TASK(Context, "getStockQuote");
226     SetupSystemBox(Context_TASK_getStockQuote);
227     SetupParam(Context_TASK_getStockQuote, Box.System, Box.Name, "getStockQuote");
228     SetupParam(Context_TASK_getStockQuote, Box.System, Box.Type, Box.Activity);
229     SetupParam(  Context_TASK_getStockQuote,
230           Box.System,
231           Box.Prog,
232           "StockQuotePort_getStockQuote");
233     SetupParam(Context_TASK_getStockQuote, Box.System, Box.MaxRestart, "0");
234     SetupParam(Context_TASK_getStockQuote, Box.Input, "symbol", "");
235     SetupParam(Context_TASK_getStockQuote, Box.Output, "Result", "");
237     TID Context_TASK_getExchangeRate = TASK(Context, "getExchangeRate");
238     SetupSystemBox(Context_TASK_getExchangeRate);
239     SetupParam(Context_TASK_getExchangeRate, Box.System, Box.Name, "getExchangeRate");
240     SetupParam(Context_TASK_getExchangeRate, Box.System, Box.Type, Box.Activity);
241     SetupParam(  Context_TASK_getExchangeRate,
242           Box.System,
243           Box.Prog,
244           "CurrencyExchangePort_getRate");
245     SetupParam(Context_TASK_getExchangeRate, Box.System, Box.MaxRestart, "0");
246     SetupParam(Context_TASK_getExchangeRate, Box.Input, "country1", "usa");
247     SetupParam(Context_TASK_getExchangeRate, Box.Input, "country2", "");
248     SetupParam(Context_TASK_getExchangeRate, Box.Output, "Result", "");
250     TID Context_TASK_Multiply = TASK(Context, "Multiply");
251     SetupSystemBox(Context_TASK_Multiply);
252     SetupParam(Context_TASK_Multiply, Box.System, Box.Name, "Multiply");
253     SetupParam(Context_TASK_Multiply, Box.System, Box.Type, Box.Activity);
254     SetupParam(Context_TASK_Multiply, Box.System, Box.Prog, "ProgramMul_JS");
255     SetupParam(Context_TASK_Multiply, Box.System, Box.MaxRestart, "0");
256     SetupParam(Context_TASK_Multiply, Box.Input, "a", "");
257     SetupParam(Context_TASK_Multiply, Box.Input, "b", "");
258     SetupParam(Context_TASK_Multiply, Box.Output, "result", "");
259   }
260 }
```

Bibliography

[1] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In VLDB 2001 [13], pages 149–158. 156

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, July 1997. 5

[3] G. Alonso. Myths around Web services. *Bulletin of the IEEE Technical Committee on Data Engineering*, 25(4):3–9, December 2002. 45

[4] G. Alonso, W. Bausch, C. Pautasso, M. Hallett, and A. Kahn. Dependable Computing in Virtual Laboratories. In ICDE2001 [264], pages 235–242. 11, 17, 32, 37

[5] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: Concepts, Architectures and Applications*. Springer, November 2003. 8, 9, 46

[6] G. Alonso, U. Fiedler, C. Hagen, A. Lazcano, H. Schuldt, and N. Weiler. WISE: Business to business e-commerce. In RIDE'99 [84], pages 132–139. 11, 17

[7] G. Alonso and C. Hagen. Geo-Opera: Workflow Concepts for Spatial Processes. In SSD97 [205], pages 238–258. 17

[8] G. Alonso and C. Hagen. Beyond the black box: event-based inter-process communication in process support systems. In ICDCS'99 [90]. 17

[9] G. Alonso, C. Hagen, H.-J. Schek, and M. Tresch. Distributed processing over stand-alone systems and applications. In VLDB 1997 [118], pages 575–579. 16

[10] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Advanced transaction models in the workflow contexts. In ICDE 1996 [219], pages 574–581. 44

[11] Altova. *Mapforce 2004*. `http://www.altova.com/products_mapforce.html`. 6

[12] Apache Software Foundation. *AXIS version 1.1*. `http://xml.apache.org/axis`. 9, 180

[13] P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors. *Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001)*, Roma, Italy, September 2001. 213, 216

[14] P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors. *Proceedings of the 5th International Conference on Advances in Database Technology (EDBT'96)*, Volume 1057 of *Lecture Notes in Computer Science*, Avignon, France, March 1996. Springer. 219

[15] V. R. Aragao and A. A. Fernandes. Conflict resolution in Web service federations. In ICWS-Europe 2003 [119], pages 109–122. 6, 65

[16] U. Assmann. *Invasive Software Composition*. Springer, 2003. ETHBIB 782616. 7

[17] M. Auguston and A. Delgado. Iterative constructs in the visual data flow language. In VL97 [228], pages 152–159. 6

[18] T. Baeyens. Java business process management. `http://www.jbpm.org/`. 45

[19] M. K. Basu. SeWeR: A customizable and integrated dynamic HTML interface to bioinformatics services. *Bioinformatics*, 17(6):577–578, 2001. 54

[20] T. Bauer and P. Dadam. A distributed execution environment for large-scale workflow management systems with subnets and server migration. In CoopIS'97 [46], pages 99–108. 16

[21] W. Bausch. *OPERA-G - A Microkernel for Computational Grids*. PhD thesis, Diss. ETH Nr. 15395, December 2003. 12, 17, 115, 121, 123, 131, 161, 166, 178, 198

[22] W. Bausch, C. Pautasso, and G. Alonso. Programming for dependability in a service based grid. In CCGrid03 [137], pages 164–171. 11, 17, 75

[23] W. Bausch, C. Pautasso, R. Schaeppi, and G. Alonso. Bioopera: Cluster-aware computing. In CLUSTER 2002 [93], pages 99–106. 17, 75, 172

[24] A. Bayucan, R. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. *Portable Batch System External Reference Specification*. MRJ Technology Solutions, May 1999. 75

[25] A. Beguelin, J. J. Dongarra, A. Geist, R. Manchek, K. Moore, R. Wade, and V. S. Sunderam. HeNCE: Graphical development tools for network-based concurrent computing. In SHPCC-92 [241], pages 129–136. 5, 31, 199

[26] P. Bernus, K. Mertins, and G. Schmidt, editors. *Handbook on Architecture of Information Systems*. Springer, Berlin, 1998. ETHBIB 778781. 11

[27] B. Bhargava. A study of communication delays for web transactions. *Cluster Computing*, 4(4):319–333, October 2001. 59

[28] D. Box. *Essential COM*. Addison Wesley, 6th edition, 2000. ETH-INFK IH.98.26. 7

[29] BPMI. *BPML: Business Process Modeling Language 1.0*. Business Process Management Initiative, Match 2001. `http://www.bpmi.org`. 9, 45, 101, 114

[30] BPMI. *BPMN: Business Process Modeling Notation 1.0*. Business Process Management Initiative, 2003. `http://www.bpmi.org`. 44

[31] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *COMPUTER*, 20(4):10–19, April 1987. 7

[32] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(1):75–83, Spring 1995. 5

[33] A. Buchmann, F. Casati, L. Fiege, M.-C. Hsu, and M.-C. Shan, editors. *Proceedings of the Third International Workshop on Technologies for E-Services (TES 2002)*, Volume 2444 of *Lecture Notes in Computer Science*, Hong Kong, China, August 2002. Springer. 221

[34] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In WWW 2003 [103], pages 403–410. 9

[35] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee. Scaling up visual programming languages. *COMPUTER*, 28(3):45–54, March 1995. 21, 147, 150

[36] C. Bussler. *B2B Integration. Concepts and Architecture*. Springer, 2002. 22

[37] A. Caliano. Automatic layout. Master's thesis, ETH Department of Computer Science, March 2003. 150

[38] S. Cannan and G. Otten. *SQL - The Standard Handbook*. Mc Graw Hill, 1993. 62

[39] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: Workflow management for grid computing. In CCGrid03 [137], pages 198–205. 11

[40] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, 1990. ETHBIB 749112. 156

[41] F. Casati and A. Discenza. Modeling and managing interactions among business processes. *Journal of Systems Integration*, 10(2):145–168, April 2001. 11

[42] F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26:143–163, 2001. 5, 8, 9, 11

[43] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Journal of Visual Languages and Computing*, 8(2):215–260, April 1997. 6

[44] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a graphical language for querying and restructuring XML documents. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada, 1999. 6

[45] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, December 2002. `http://www.webml.org`. 5, 9

[46] A. L. P. Chen, W. Klas, and M. P. Singh, editors. *Proceedings of the 2nd IFCIS International Conference on Cooperative Information Systems (CoopIS'97)*, Kiawah Island, South Carolina, USA, June 1997. 214

[47] J.-Y. Chung, K.-J. Lin, and R. G. Mathieu. Web services computing–advancing software interoperability. *COMPUTER*, 36(10):35–37, October 2003. 9

[48] J. Church and N. Gandal. Network effects, software provision and standardization. *Journal of Industrial Economics*, 40(1):85–103, 1992. 7

[49] Collaxa. BPEL Server and Designer. `http://www.collaxa.com`. 15

[50] K. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004. ETHBIB 783640. 121, 144

[51] G. Copeland and D. Maier. Making smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 316–325, Boston, Massachusetts, 1984. 9

[52] B. J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6):25–33, 1990. 7

[53] P. T. Cox, F. R. Giles, and T. Pietrzykowski. *Prograph*, In Visual object-oriented programming: concepts and environments, chapter 3, pages 45–66. Manning Publications Co., 1995. 5

[54] P. T. Cox and B. Song. A formal model for component-based software. In HCC 2001 [140], pages 304–311. 7

[55] M. Crispin. *Internet Message Access Protocol*. IETF RFC 2060, 1996. 79

[56] J. Crupi, D. Malks, and D. Alur. *Core J2EE Patterns: Best practices and design strategies*. Prentice Hall, 2nd edition, 2003. 7

[57] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 22(12):827–850, September 2001. 15

[58] F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana. Exception handling in the BPEL4WS language. In BPM2003 [239], pages 276–290. 13, 26

[59] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM symposium on Operating systems principles*, pages 202–215, Banff, Alberta, Canada, 2001. 156

[60] U. Dayal, M. Hsu, and R. Ladin. Business process coordination: State of the art, trends, and open issues. In VLDB 2001 [13], pages 3–13. 12

[61] H. M. Deitel, P. J. Deitel, B. DuWaldt, and L. K. Trees. *Web services: a Technical Introduction*. Prentice Hall, 2003. HES-EIF 681.351 WEB B 02-593. p. 12. 8

[62] J. B. Dennis. Dataflow supercomputers. *COMPUTER*, 13(11):48–56, 1980. 6, 44

[63] E. di Nitto, L. Lavazza, M. Schiavoni, E. Tracanella, and M. Trombetta. Deriving executable process descriptions from UML. In ICSE 2002 [230], pages 155–165. 10, 22

[64] A. Dogac, L. Kalichenko, M. T. Ozsu, and A. Sheth, editors. *Workflow Management Systems and Interoperability*, Volume 164 of *Computer and Systems Sciences*. NATO ASI Series, 1997. 11, 220, 221

[65] V. Draluk. Discovering Web services: An Overview. In VLDB 2001 [13], pages 637–640. 9

[66] ebXML. *ebXML Business Process Specification Schema (BPSS) 1.01*, 2001. `http://www.ebxml.org/specs/ebBPSS.pdf`. 9

[67] C. A. Ellis. Information control nets: A mathematical model of office information flow. In *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240, Boulder, CO, 1979. 10

[68] D. W. Embley and W. Y. Mok. Developing XML documents with guaranteed "good" properties. In ER 2001 [132], pages 426–441. 151

[69] W. Emmerich. Distributed component technologies and their software engineering implications. In ICSE 2002 [230], pages 537–546. 7

[70] Entigen. *BioNavigator*. `www.bionavigator.com`. 54

[71] M. Erwig. Xing: A visual XML query language. *Journal of Visual Languages and Computing*, 14(1):5–45, February 2003. 6

[72] O. Etzion and P. Scheuermann, editors. *7th International Conference on Cooperative Information Systems (CoopIS-2000)*, Volume 1901 of *Lecture Notes in Computer Science*, Eilat, Israel, September 2000. Springer. 225

[73] D. Fensel and C. Bussler. The web service modeling framework wsmf. *Electronic Commerce Research and Applications*, 1(2):113–137, Summer 2002. 8

[74] D. Florescu, A. Gruenhagen, and D. Kossmann. Xl: An XML programming language for web service specification and composition. In WWW2002 [134], pages 65–76. 9

[75] D. Florescu, A. Gruenhagen, and D. Kossmann. Xl: A platform for Web services. In CIDR 2003 [218]. 9

[76] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces: Principles, Patterns and Practice*. Addison Wesley, 1999. 156

[77] J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors. *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003)*, Berlin, Germany, September 2003. 225

[78] A. Fuggetta. Software process: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000) - Future of Software Engineering Track*, pages 25–34, Limerick, Ireland, June 2000. 11

[79] A. Fukunaga, W. Pree, and T. D. Kimura. Functions as objects in a data flow based visual language. In *Proceedings of the 1993 ACM conference on Computer science*, pages 215–220, Indianapolis, IN, February 1993. 6

[80] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins.* Addison Wesley, October 2003. 152

[81] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996. 151

[82] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992. 7, 9, 43, 44

[83] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modelling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995. 5, 9, 11, 119

[84] D. Georgakopoulos and L. Maciaszek, editors. *Proceedings of the 9th IEEE International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDE-VE'99)*, Sidney, Australia, March 1999. 213

[85] A. Geppert and D. Tombros. Event-based distributed workflow execution with EVE. Technical Report 96.05, Dept. of Computer Science, University of Zurich, 1998. 15

[86] G. Gonnet, M. Hallett, C. Korostensky, and L. Bernardin. Darwin version 2.0: An interpreted computer language for the biosciences. *Bioinformatics*, 16:101–103, 2000. 62

[87] G. H. Gonnet, T. F. Jenny, and L. J. Knecht. *The Computational Biochemistry Server at ETHZ*, 2002. `http://cbrg.inf.ethz.ch/Server/`. 54

[88] Google. *Google Web APIs*. `http://api.google.com/GoogleSearch.wsdl`. 67

[89] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002. 8, 45

[90] M. C. Gouda, editor. *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, Austin, Texas, USA, June 1999. 213

[91] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. Merging the cca component model with the ogsi framework. In CCGrid03 [137], pages 182–189. 5, 8

[92] P. Grefen and R. R. de Vries. A reference architecture for workflow management systems. *Journal of Data and Knowledge Engineering*, 27(1):31–57, 1998. 15, 119

[93] B. Gropp, R. Pennington, D. Reed, M. Baker, M. Brown, and R. Buyya, editors. *Proceedings of the 2002 IEEE International Conference on Cluster Computing (CLUSTER 2002)*, Chicago, IL, USA, September 2002. 214

[94] J. C. Grundy, R. Mugridge, J. G. Hosking, and P. Kendall. A visual language and environment for EDI message translation. In HCC 2001 [140], pages 330–331. 7

[95] X. Gu, K. Nahrstedt, R. N. Chang, and C. Ward. QoS-assured service composition in managed service overlay networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 194–201, 2003. 16

[96] C. A. Gurr. Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. *Journal of Visual Languages and Computing*, 10(4):317–342, 1999. 22

[97] C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10), October 2000. 17, 44

[98] C. J. Hagen. *A Generic Kernel for Reliable Process Support*. PhD thesis, Diss. ETH Nr. 13114, March 1999. 11, 12, 16, 49, 87, 97, 119, 123, 131

[99] J. Hahn and J. Kim. Why are some representations (sometimes) more effective? In *Proceeding of the 20th international conference on Information Systems*, pages 245–259, Charlotte, North Carolina, United States, 1999. 44

[100] D. Harel. Statecharts: A visual formalism for complex system. *Science of Computer Programming*, 8(3):231–274, 1987. 6, 21

[101] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering - Putting the Pieces Together*. Addison Wesley, 2001. EPF BC CG 4354. 7

[102] P. Heinl and H. Schuster. Towards a highly scaleable architecture for workflow management systems. In *Proceedings of the 7th International Workshop on Database and Expert Systems Applications*, pages 439–444, 1996. 15

[103] G. Hencsey and B. White, editors. *Proceedings of the Twelfth International World Wide Web Conference*, Budapest, Hungary, May 2003. 214

[104] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3(1):69–101, 1992. 6, 21, 43, 44, 102

[105] J. Hosking and P. Cox, editors. *Proceedings of the 2003 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2003)*, Auckland, New Zealand, October 2003. 222

[106] IBM. BPEL4WS Java Runtime. `http://www.alphaworks.ibm.com/tech/bpws4j`. 15

[107] IBM. *Emerging Technologies Toolkit (ETTK)*. `http://www.alphaworks.ibm.com/tech/ettk`. 9

[108] IBM. *TSpaces*. `http://www.almaden.ibm.com/cs/Tspaces/`. 184

[109] IBM. Autonomic Computing: Special Issue. *IBM Systems Journal*, 42(1), 2003. 17, 161

[110] IBM and Apache Foundation. *Web Service Invocation Framework (WSIF)*, 2003. `http://ws.apache.org/wsif/`. 9, 39

[111] IBM and BEA Systems. *BPELJ: BPEL for Java technology*, March 2004. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpelj/`. 13, 60

[112] IBM, Microsoft, and BEA Systems. *Business Process Execution Language for Web services (BPEL4WS) 1.0*, August 2002. `http://www.ibm.com/developerworks/library/ws-bpel`. 9, 12, 13, 15, 45, 60, 83, 101, 114, 121, 125

[113] IBM, Microsoft, and BEA Systems. *Web services Coordination (WS-Coordination)*, 2002. `http://www.ibm.com/developerworks/library/ws-coor`. 9

[114] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. Fabrik: a visual programming environment. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'88)*, pages 176–190, San Diego, CA, 1988. 5

[115] Ivyteam. Process modeling software. `http://www.ivyteam.com`. 17

[116] S. Iyengar. Business process integration using UML and BPEL4WS. In M. Glinz and H.-P. Hoidn, editors, *Components: the Future of Software Engineering? (SI-SE 2004)*, Zurich, Switzerland, March 2004. 13, 21, 22

[117] A. Jackobs and F. Titsworth, editors. *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, USA, May 2003. 220, 221, 224

[118] M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors. *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97)*, Athens, Greece, August 1997. Morgan Kaufmann. 213

[119] M. Jeckle and L.-J. Zhang, editors. *Proceedings of the International Conference on Web services (ICWS-Europe 2003)*, Volume 2853 of *Lecture Notes in Computer Science*, Erfurt, Germany, September 2003. Springer. 213, 221, 226

[120] C. Jensen. *Temperature Conversion Service*. `http://developerdays.com/cgi-bin/tempconverter.exe/wsdl/ITempConverter`. 179

[121] T. Jeweel and D. Chappell. *Java Web services*. O'Reilly, 2002. 9

[122] L. jie Jin, F. Casati, M. Sayal, and M.-C. Shan. Load balancing in distributed workflow management system. In *Proceedings of the ACM Symposium on Applied Computing*, pages 522–530, 2001. 16

[123] S. Joosten. Why modelers wreck workflow innovations. In BPM2000 [238], pages 289–300. 10

[124] M. Kamath, G. Alonso, R. Guenthoer, and C. Mohan. Providing high availability in very large workflow management systems. In EDBT'96 [14], pages 427–442. 15, 119, 129, 186, 189

[125] B. W. Kernighan and P. J. Plauger. *Software Tools*. Addison Wesley, 1976. ETHBIB 727584. 58

[126] J. D. Kiper, E. Howard, and C. Ames. Criteria for evaluation of visual programming languages. *Journal of Visual Languages and Computing*, 8(2):175–192, April 1997. 5

[127] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained - the Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003. ETHBIB 783375. 12, 143, 151

[128] R. Konopka. *Developing Custom Delphi Components*. Coriolis Group Books, 2nd edition, 1996. 1, 7

[129] A. Krause, P. Nicodeme, E. Bornberg-Bauer, M. Rehmsmeier, and M. Vingron. WWW access to the SYSTERS protein sequence cluster set. *Bioinformatics*, 15:262–263, 1999. 54

[130] N. Krishnakumar and A. Sheth. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995. 132

[131] J. Kubiatowicz. Extracting guarantees from chaos. *Communications of the ACM*, 46(2):33–38, 2003. 156

[132] H. S. Kunii, S. Jajodia, and A. Sølvberg, editors. *20th International Conference on Conceptual Modeling (ER2001)*, Volume 2224 of *Lecture Notes in Computer Science*, Yokohama, Japan, November 2001. Springer. 216

[133] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991. 112

[134] D. Lassner, D. De Roure, and A. Iyengar, editors. *Proceedings of the 11th international conference on World Wide Web (WWW'02)*, Honolulu, Hawaii, USA, May 2002. 216

[135] P. Lawrence, editor. *Workflow Handbook*. Wiley, 1997. 10

[136] A. Lazcano and G. Alonso. Process based e-services. In *Proceedings of the Second International Workshop on Electronic Commerce (WELCOM 2001)*, Volume 2232 of *Lecture Notes in Computer Science*, pages 1–10, Heidelberg, Germany, November 2001. Springer. 17

[137] S. Lee, S. Sekiguschi, S. Matsuoka, and M. Sato, editors. *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid03)*, Tokyo, Japan, May 2003. 214, 217

[138] T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472, March 2001. 156, 187

[139] C. Letondal. A Web interface generator for molecular biology programs in Unix. *Bioinformatics*, 17(1):73–82, 2001. 54, 150

[140] S. Levialdi, editor. *Proceedings of the 2001 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2001)*, Stresa, Italy, September 2001. 215, 217, 222, 226

[141] F. Leymann. Web services and their composition. *Lecture Notes in Computer Science*, 2077:1–2, 2001. 9

[142] F. Leymann. *Web services Flow Language (WSFL 1.0)*. IBM, 2001. 13

[143] F. Leymann. Web services: Distributed applications without limits. In BPM2003 [239], pages 123–145. 9

[144] F. Leymann and D. Roller. Business process management with flowmark. In *Proceedings of the 39th IEEE Computer Society International Conference (CompCon '94)*, pages 230–234, February 1994. 49, 101

[145] F. Leymann and D. Roller. Building A robust workflow management system with persistent queues and stored procedures. In ICDE98 [265], pages 254–258. 131

[146] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 1999. 11

[147] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002. 11

[148] F. Leymann, H. J. Scheck, and G. Vossen. *Transactional Workflows*. Dagstuhl Seminar 9629, 1996. 11, 44

[149] B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. 7

[150] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—A hunter of idle workstations. In *Proceedings of the 8th Int'l Conf. on Distributed Computing Systems*, pages 104–111, 1988. 75

[151] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 147–155, Orlando, FL, October 1987. 6, 34

[152] J. Marks, editor. *Proceedings of the 8th International Symposium on Graph Drawing (GD 2000)*, Volume 1984 of *Lecture Notes in Computer Science*, Colonial Williamsburg, VA, USA, September 2000. Springer. 150

[153] R. McClatchey, J.-M. L. Goff, N. Baker, W. Harris, and Z. Kovacs. A distributed workflow and product data management application for the construction of large scale scientific apparatus. In Nato ASI Series, vol. 164 [64], pages 18–34. 11

[154] M. D. McIlroy. Mass-produced software components. In WCSE'68 [170], pages 138–150. 1, 7

[155] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing Web services on the semantic web. *The VLDB Journal*, 12(4):333–351, November 2003. 9

[156] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000. 5, 8

[157] J. Meidanis, G. Vossen, and M. Weske. Using workflow management in DNA sequencing. In *Proceedings of the 1st International Conference on Cooperative Information Systems (CoopIS96) Brussels, Belgium*, June 1996. 11

[158] D. G. Messerschmitt and C. Szyperski. *Software Ecosystems: Understanding an Indispensable Technology and Industry*. MIT Press, 2003. ETHBIB 783612. 7, 12

[159] B. Meyer. *Object Oriented Software Construction. Second Edition*. Prentice Hall, 1997. 7

[160] B. Meyer. The grand challenge of trusted components. In ICSE2003 [117], pages 660–667. 7

[161] Microsoft Corp. *Web services Enhancements for Microsoft .NET (WSE) 2.0.* `http://msdn.microsoft.com/webservices/building/wse/`. 9

[162] R. Milner. *Communicating and mobile systems: the π calculus*. Cambridge University Press, 1999. ETHBIB 775144. 14, 44, 200

[163] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995. 150

[164] C. Mohan. Recent trends in workflow management products, standards and research. In Nato ASI Series, vol. 164 [64], pages 396–409. 11

[165] C. Mohan. Dynamic e-business: Trends in Web services. In TES 2002 [33], pages 1–5. 9

[166] M. Mosconi and M. Porta. Iteration constructs in data-flow visual programming languages. *Computer Languages*, 26(2–4):67–104, July 2000. 6, 31

[167] M. Muench and A. Schuerr. Leaving the visual language ghetto. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 148–155, 1999. 5, 9, 21

[168] P. Muth, D. Wodtke, J. Weissenfels, A. Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems*, 10(2):159–184, 1998. 15, 119

[169] J. Myers and M. Rose. *Post Office Protocol - version 3*. IETF RFC 1939, 1996. 79

[170] P. Naur and B. Randell, editors. *Proceedings of the Working Conference on Software Engineering*, Garmisch-Partenkirchen, Germany, October 1968. NATO Science Committee. 220

[171] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992. 62

[172] F. Nordsieck. *Grundlagen der Organisationshlehre*. Poeschel, Stuttgart, 1934. ETHHDB 922941. 10

[173] Oasis. *Universal Description, Discovery and Integration of Web services (UDDI) Version 3.0*, 2002. `http://uddi.org/pubs/uddi_v3.htm`. 8

[174] J. Oberleitner and S. Dustdar. Constructing Web services out of generic component compositions. In ICWS-Europe 2003 [119], pages 37–48. 8, 46

[175] J. Oberleitner, T. Gschwind, and M. Jazayeri. The vienna component framework enabling composition across component models. In ICSE2003 [117], pages 25–35. 8, 103

[176] Object Management Group. *Unified Modeling Language (UML)*, 2004. `http://www.uml.org`. 87

[177] Object Management Group (OMG). *CORBA: Common Object Request Broker Architecture*. `http://www.corba.org/`. 7

[178] B. Omelayenko and D. Fensel. A two-layered integration approach for product information in B2B e-commerce,. In K. Bauknecht, S.-K. Madria, and G. P. (eds.), editors, *Proceedings of the Second International Conference on Electronic Commerce and Web Technologies (EC WEB-2001)*, Volume 2115 of *LNCS*, pages 226–239, Munich, Germany, September 2001. 73

[179] OpenStorm. Service Orchestrator, February 2004. `http://www.openstorm.com`. 15

[180] OSGi Alliance. *Open Services Gateway Interface Service Platform Specification v3*, March 2003. `http://www.osgi.org`. 7

[181] L. J. Osterweil. Software processes are software too. In *Proceedings of the 9th International Conference on Software Engineering*, pages 2–13, Washington, DC, 1987. IEEE Computer Society Press. 11

[182] J. K. Ousterhour. Scripting: Higher level programming for the 21st century. *COMPUTER*, 31(3):23–30, March 1998. 62

[183] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, October 2003. 8, 9

[184] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. 7

[185] C. Pautasso. JOpera: Process Support for Web services. `http://www.iks.ethz.ch/jopera/download`. 16

[186] C. Pautasso. Resource Management and Scheduling for the BioOpera Process Support System. Master's thesis, Politecnico di Milano, April 2000. 17

[187] C. Pautasso and G. Alonso. Visual composition of Web services. In HCC 2003 [105], pages 92–99. 6, 21, 87, 147

[188] L. Perrochon, G. Wiederhold, and R. Burback. A compiler for composition: CHAIMS. pages 44–51, June 1997. 7

[189] Persistence of Vision Development Team. *POV-Ray 3.1*, 2002. `http://www.povray.org`. 78

[190] J. L. Peterson. Petri Nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977. 6, 14, 21, 44

[191] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995. 5, 21, 147

[192] E. Pietriga and J.-Y. Vion-Dury. VXT: Visual XML transformer. In HCC 2001 [140], pages 404–405. 6

[193] J. B. Postel. *Simple Mail Transfer Protocol*. IETF RFC 821, 1982. 79

[194] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. *ACM SIGCOMM Computer Communication Review*, 29(4):15–25, 1999. 173

[195] A. Ran. Software isn't built from lego blocks. In *Proceedings of the ACM Symposium on Software Reusability (SSR)*, pages 164–169, 1999. 7

[196] J. Raskin. *The Humane Interface: New directions for designing interactive systems*. Addison-Wesley, 2000. ETHBIB 779680. 5, 23

[197] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998. ETHBIB 774289. 5

[198] C. Rupp. Building an Application Server for BioOpera. Master's thesis, ETH Department of Computer Science, January 2003. 150

[199] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997. ETHBIB 771777. 7

[200] T. Schael. *Workflow Management Systems for Process Organisations*. LNCS 1096. Springer, Berlin, 1996. 11

[201] A. Schill and C. Mittasch. Workflow management systems on top of OSF DCE and OMG CORBA. *Distributed Systems Engineering*, 3(4):250–262, 1996. 119

[202] A. Schmidt, T. Sindt, M. Tepegoez, and G. Joeris. FlowTEC - an information system supporting virtual enterprises. In *Proceedings of the 2nd International Conference on Concurrent Multidisciplinary Engineering (CME'99)*, Bremen, 1999. 11

[203] J.-G. Schneider. *Components, Scripts, and Glue: A Conceptual Framework for Software Composition*. PhD thesis, Universität Bern, 1999. 8, 62

[204] B. Schneier. *Applied Cryptography*. Wiley, 1994. ETHBIB 758537. page 242. 173

[205] M. Scholl and A. Voisard, editors. *Proceedings of the 5th International Symposium on Advances in Spatial Databases, SSD'97*, Volume 1262 of *Lecture Notes in Computer Science*, Berlin, Germany, July 1997. Springer. 213

[206] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and isolation for transactional processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, 2002. 11, 17, 44

[207] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek. Peer to peer process execution with OSIRIS. In *Proceedings of the Service-Oriented Computing Conference (ICSOC 2003)*, pages 483–498, 2003. 16

[208] H. Schuster and P. Heinl. A workflow data distribution strategy for scalable workflow management systems. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages 174–176, 1997. 15, 186, 189

[209] H. Schuster, S. Jablonski, P. Heinl, and C. Bussler. A general framework for the execution of heterogeneous programs in workflow management systems. In *Proceedings of the 1st IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*, pages 104–113, Los Alamitos, CA, 1996. IEEE Computer Society Press. 11, 48

[210] M. Senger, T. Flores, K. Glatting, P. Ernst, A. Hotz-Wagenblatt, and S. Suhai. W2H: WWW interface to the GCG sequence analysis package. *Bioinformatics*, 14(5):452–457, 1998. 54

[211] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of 21st International Computer Software and Applications Conference*, pages 6–13, Washington, D.C., August 1997. 7

[212] A. P. Sheth. *Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics*, In Interoperating Geographic Information Systems, pages 5–30. Kluwer Academic Publishers, 1998. 65, 70

[213] B. A. Shirazi, A. R. Hurson, and K. M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995. 172

[214] H. A. Simon. *The Sciences of the Artificial*. MIT Press, 3rd edition, 1996. ETHBIB 959423. 1

[215] H. Smith. Business process management, the third wave: business process modelling language BPML and its *pi*-calculus foundations. *Information and Software Technology*, 45(15):1065–1069, December 2003. 11

[216] H. Smith. *Enough is enough in the field of BPM: We don't need BPELJ: BPML semantics are just fine*, April 2004. `http://www.bpm3.com/bpelj/BPELJ-Enough-Is-Enough.pdf`. 13

[217] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003. 156

[218] M. Stonebraker and D. Dewitt, editors. *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, USA, January 2003. 216

[219] S. Y. W. Su, editor. *Proceedings of the 12th International Conference on Data Engineering (ICDE 1996)*, New Orleans, Louisiana, February 1996. 213, 225

[220] Sun Microsystems. *Java Message Service API version 1.1*. `http://java.sun.com/products/jms/`. 79, 80, 182

[221] Sun Microsystems. *Java Web services Developer Pack 1.3*. `http://java.sun.com/webservices`. 9

[222] SUN microsystems. *Sun Grid Engine*. `http://www.sun.com/software/gridware/`. 75

[223] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. In *AFIPS Conference Proceedings 23*, pages 323–328, 1963. 5

[224] C. Szyperski. *Component Software - Beyond Object Oriented Programming*. Addison Wesley, 2nd edition, 2002. 7

[225] C. Szyperski. Component technology - what, where, and how? In ICSE2003 [117], pages 684–693. 1, 8, 85

[226] S. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 2(2):127–139, June 1990. 5, 144

[227] S. Thatte. *XLANG: Web services for Business Process Design*. Microsoft, May 2000. 9, 13

[228] G. Tortora, editor. *Proceedings of the 1997 IEEE Symposium on Visual Languages*, Capri, Italy, September 1997. 213

[229] W. Tracz. *Confessions of a Used Program Salesman*. Addison-Wesley, 1995. ETHBIB 764161. 1, 5

[230] W. Tracz, editor. *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, Orlando, FL, USA, May 2002. 216

[231] E. R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990. ETHBIB 750940. 5

[232] R. Valdes. Introducing interoperable objects. *Dr. Dobbs Journal*, 19(16):4–6, May 1994. `http://www.ddj.com/articles/1994/9416/`. 7

[233] W. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003. 11, 13, 22, 31, 40, 44, 101

[234] W. M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. 6, 21, 44

[235] W. M. P. van der Aalst. Process-oriented architectures for electronic commerce and interorganizational workflow. *Information Systems*, 24(8):639–671, December 1999. 11

[236] W. M. P. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–85, 2003. 8, 14

[237] W. M. P. van der Aalst and P. J. S. Berens. Beyond workflow management: Product-driven case handling. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, pages 42–51, 2001. 11

[238] W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management, Models, Techniques, and Empirical Studies*, Volume 1806 of *Lecture Notes in Computer Science*. Springer, 2000. 219

[239] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske, editors. *Proceedings of the International Conference on Business Process Management (BPM 2003)*, Volume 2678 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 2003. Springer. 10, 11, 215, 220

[240] G. van Rossum and F. L. Drake. *The Python Language Reference Manual*. Network Theory Ltd, September 2003. `http://www.python.org`. 62

[241] R. Voigt, J. Saltz, and L. O'Connor, editors. *Proceedings of the 1992 Scalable High Performance Computing Conference (SHPCC-92)*, Williamsburg, Virginia, April 1992. 214

[242] G. M. Vose and G. Williams. Labview: Laboratory virtual instrument engineering workbench. *Byte*, 11(9):84–92, September 1986. 5

[243] W3C. *Extensible Stylesheet Language Transformations (XSLT) 1.0*, 1999. http://www.w3.org/TR/xslt. 6, 65, 67

[244] W3C. *XML Path Language (XPath) 1.0*, 1999. http://www.w3.org/TR/xpath. 65, 67

[245] W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. http://www.w3.org/TR/SOAP. 8, 50

[246] W3C. *Web services Definition Language (WSDL) 1.1*, 2001. http://www.w3.org/TR/wsdl. 8, 50

[247] W3C. XML Schema, 2001. http://www.w3.org/TR/xmlschema-0/. 65, 70, 87, 201

[248] W3C. *Web services Choreography Interface (WSCI) 1.0*, 2002. http://www.w3.org/TR/wsci. 9

[249] W3C. *Web services Conversation Language (WSCL) 1.0*, 2002. http://www.w3.org/TR/wscl10. 9

[250] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 2000. http://www.perl.org. 62

[251] R. Weber, C. Schuler, H. Schuldt, H.-J. Schek, and P. Neukomm. Web Service Composition with O'GRAPE and OSIRIS. In VLDB2003 [77], pages 1081–1084. 16

[252] M. Weske, G. Vossen, and C. Medeiros. Scientific workflow management: WASA architecture and applications. Technical Report 03/96-I, Universitat Munster, 1996. 11

[253] WFMC. The workflow reference model. Technical Report WFMC-TC-1003, Workflow Management Coalition, Bruxelles, Belgium, January 1995. 119

[254] G. Wiederhold, P. Wegner, and S. Ceri. Towards megaprogramming: A paradigm for component-based programming. *Communications of the ACM*, 35(11):89–99, 1992. 5, 7

[255] N. Wirth. On the design of programming languages. In *Proceedings of the Information Processing Congress (IFIP 74)*, pages 386–393, Stockholm, Sweden, 1974. 2, 4

[256] G. Wirtz, M. Weske, and H. Giese. Extending UML with workflow modeling capabilities. In CoopIS 2000 [72], pages 30–41. 5, 6, 10, 21, 44

[257] D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz-Dittrich. The mentor project: Steps toward enterprise-wide workflow management. In ICDE 1996 [219], pages 556–565. 6, 15, 21

[258] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Pattern-based analysis of BPEL4WS. Technical Report FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002. 12, 13

[259] L. Wood, A. Hors, V. Apparao, L. Cable, M. Champion, J. Kesselman, P. Hegaret, T. Pixley, J. Robie, P. Sharpe, and C. Wilson. Document Object Model (DOM), 1999. http://www.w3.org/TR/DOM-Level-2/. 151

[260] W. Wulf and M. Shaw. Global variable considered harmful. *ACM SIGPLAN Notices*, 8(2):28–34, February 1973. 102

[261] XMethods. *Currency Exchange Rate Service*. http://www.xmethods.net/sd/2001/CurrencyExchangeService.wsdl. 55

[262] XMethods. *Delayed Stock Quote Service*. http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl. 55

[263] J. Yang and M. P. Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, April 2004. 9

[264] D. C. Young, editor. *Proceedings of the 17th International Conference on Data Engineering (ICDE2001)*, Heidelberg, Germany, April 2001. 213

[265] P. Yu, editor. *Proceedings of the 14th International Conference on Data Engineering (ICDE 98)*, Orlando, Florida, USA, February 1998. 220

[266] J. Zawinski. *Unity of Interface*, 1998. `http://www.mozilla.org/unity-of-interface.html`. 85

[267] U. Zdun, M. Voelter, and M. Kircher. Design and implementation of an asynchronous invocation framework for Web services. In ICWS-Europe 2003 [119], pages 64–78. 39

[268] K. Zhang, D.-Q. Zhang, and Y. Deng. A visual approach to XML document design and transformation. In HCC 2001 [140], pages 312–319. 6

[269] L.-J. Zhang and M. Jeckle. The next big thing: Web services collaboration. In ICWS-Europe 2003 [119], pages 1–10. 8, 9

[270] M. D. Zinsman. *Representation, Specification and Automation of Office Procedures*. PhD thesis, University of Pennsylvania, Warton School of Business, 1977. 10

Index

Abstract, 95
AccessMethod, 113
Activity, 23, 100
Architecture, 121, 147, 150, 154
Arrow, 107

Binding, 104
BioOpera, 17, 76
Box, 109
`BPEL`, 13, 45, 83, 114
`BPML`, 13, 45, 114

Comments, 33
Compiler, 119
Component Type Modeling, 47, 113
ComponentType, 113
Conditions, 25
Control Flow, 6, 25

Data Flow, 6, 23, 48, 101

Echo, 73
email, 80
Exception Handling, 26

Flexibility, 160, 163, 178

GroupBox, 108

`HTTP`, 54

Interface, 97
Iteration, 6, 31

Java
    Java Scripts, 60, 180
    Mapping, 129
    Method Calls, 60, 180
    Programs, 61
JMS, 80

JOpera Visual Composition Language (JVCL), 21, 105

Late Binding, 36

Model Driven Architecture, 12, 143

NamedObject, 96

Object, 95
`OCR`, 87, 123
`OML`, 87

Parameter, 103
Peer to Peer, 156, 176
Portable Batch System, 75
Process, 23, 97
    Instantiation, 130, 188
    Modeling Language, 11
    Navigation, 129
      Parallel, 162
    Plugin, 137
    State, 136
Program, 112
Program Library, 112

Recursion, 33
RefBox, 110
Reflection, 6, 34
Reliability, 186
Root, 94

Scalability, 188
Service, 8
Shell Commands, 58
`SOAP`, 50
Software Composition, 7
`SQL`, 63
SubProcess, 23, 74, 101
Synchronization, 26

System Parameters, 48

Task, 23, 97
    State, 25, 132, 133
TextBox, 110

View, 106
ViewObject, 107
Visual Adaptation, 6, 57, 69, 183
Visual Programming Languages, 5
Visual Scalability, 150

Web Services, 8, 45, 50, 180
Workflow, 10, 84
    Patterns, 12

`XML`, 6, 65
`X-Path`, 67
`XSLT`, 67

Curriculum Vitae: Cesare Pautasso

9.12.1975	Born in Como, Italy

Education

5.1993	Carroll High School Diploma, Southlake, Texas (USA)
7.1994	Maturita' scientifica (60/60) Liceo Scientifico "Giovio", Como, Italy
9.1994–4.2000	Undergraduate studies Computer Science Engineering at Politecnico di Milano
9.1999–3.2000	ERASMUS exchange program with ETH Zürich
4.2000	Laurea in Ingegneria Informatica (100/100 cum laude) from Politecnico di Milano
6.2000–	Doctorate Department of Computer Science of ETH Zürich

Work Experience

7.1997–8.1997	PC Support Analyst at
7.1998–9.1998	Prudential-Bache International (UK) Ltd (9 Devonshire Square, London EC2M 4HP)
6.2000–	Research assistant at the Department of Computer Science of ETH Zürich