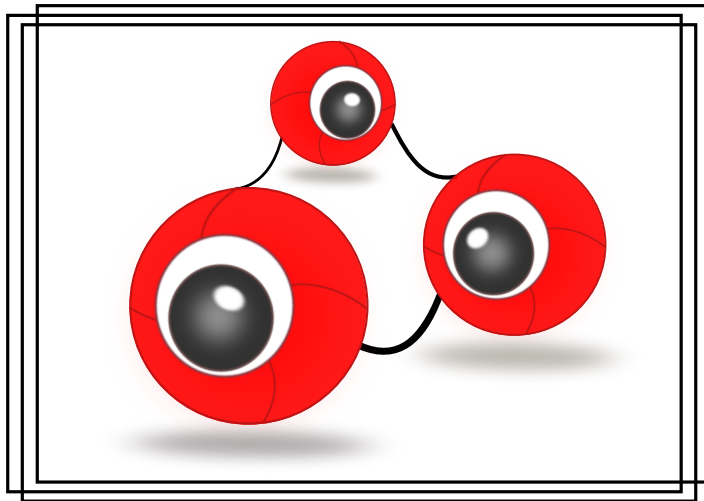




Enterprise Application Integration

**An Easy-to-Maintain
Model-Driven Engineering
Approach**



Rafael Zancan Frantz

Supervised by Dr. Rafael Corchuelo

ENTERPRISE APPLICATION INTEGRATION



AN EASY-TO-MAINTAIN MODEL-DRIVEN
ENGINEERING APPROACH

RAFAEL ZANCAN FRANTZ

UNIVERSITY OF SEVILLE

DOCTORAL DISSERTATION
SUPERVISED BY DR. RAFAEL CORCHUELO



FEBRUARY, 2012

First published in February 2012 by
The Distributed Group
ETSI Informática
Avda. de la Reina Mercedes s/n
Sevilla, 41012. SPAIN

Copyright © MMXII Rafael Zancan Frantz
<http://www.tdg-seville.info>
contact@tdg-seville.info

In keeping with the traditional purpose of furthering science, education and research, it is the policy of the publisher, whenever possible, to permit non-commercial use and redistribution of the information contained in the documents whose copyright they own. You however are *not allowed* to take money for the distribution or use of these results except for a nominal charge for photocopying, sending copies, or whichever means you use redistribute them. The results in this document have been tested carefully, but they are not guaranteed for any particular purpose. The publisher or the holder of the copyright do not offer any warranties or representations, nor do they accept any liabilities with respect to them.

Classification (ACM 1998): D.1.2 [Automatic Programming]; D.1.3 [Concurrent Programming]; D.2.6 [Programming Environments]: Integrated environment, Programmer workbench; D.2.10 [Design]: Representation; D.2.11 [Software Architectures]: Domain-specific architectures, Languages, Patterns, Service-oriented architecture; D.2.12 [Interoperability]: Distributed objects; D.2.13 [Reusable Software]: Domain engineering; H.3.4 [Systems and Software]: Distributed systems; J.6 [Computer-Aided Engineering]: Computer-aided design.

Support: PhD scholarship granted by the Evangelischer Entwicklungsdienst e.V. (EED). Additional support for research visit granted by the University of Seville Research Programme and Intelligent Integration Factory, S.L, and for attending conferences by the Spanish and the Andalusian R&D&I programmes (grants TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2010-21744, TIN2008-04718-E, TIN2010-09809-E, TIN2010-10811-E, TIN2010-09988-E, and TIN2008-04951-E).

University of Seville

The committee in charge of evaluating the dissertation presented by Rafael Zancan Frantz in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering, hereby recommends _____ of this dissertation and awards the author the grade _____.

Miguel Toro Bonilla
Catedrático de Universidad
Univ. de Sevilla

Mario Gerardo Piattini Velthuis
Catedrático de Universidad
Univ. de Castilla–La Mancha

Nieves Rodríguez Brisaboa
Catedrática de Universidad
Univ. de A Coruña

Carlos Molina Jiménez
Research Associate
Newcastle University

José Luis Arjona Fernández
Contratado Doctor
Univ. de Huelva

To put record where necessary, we sign minutes in _____,
_____.



Enterprise Application Integration by Pedrinho, aged seven.

To Fabi.
To my parents.
To my brother and sister.

Contents

Acknowledgements	xv
Abstract	xvii
Resumen	xix
Resumo	xxi
Abstrakt	xxiii

I Preface

1 Introduction	3
1.1 Research context	4
1.2 Research rationale	6
1.2.1 Hypothesis	6
1.2.2 Thesis	7
1.3 Collaborations	8
1.4 Summary of contributions	10
1.5 Structure of this dissertation	13
2 Motivation	15
2.1 Introduction	16
2.2 Problems	17
2.3 Analysis of current solutions	20

2.4	Discussion	23
2.5	Our proposal	24
2.6	Summary	27

II Background Information

3	Enterprise Integration Patterns	31
3.1	Introduction	32
3.2	Categories of patterns	32
3.3	An example	35
3.4	Summary	36
4	Camel	37
4.1	Introduction	38
4.2	Exchanges	39
4.3	Endpoints	40
4.4	Processors	40
4.5	Routes	41
4.6	Error detection	42
4.7	The Café integration solution	42
4.8	Summary	44
5	Spring Integration	45
5.1	Introduction	46
5.2	Messages	47
5.3	Endpoints	47
5.4	Message channels	49
5.5	Error detection	49
5.6	The Café integration solution	50
5.7	Summary	51
6	Mule	53
6.1	Introduction	54
6.2	Messages	55
6.3	Endpoints	55

6.4	Processors	56
6.5	Flows	57
6.6	Error detection	58
6.7	The Café integration solution	58
6.8	Summary	59
7	Model-Driven Engineering	61
7.1	Introduction	62
7.2	Model-Driven Architecture	63
7.3	Software Factories	65
7.4	Summary	67

III Our Approach

8	Domain-Specific Language	71
8.1	Introduction	72
8.2	Abstract syntax	75
8.2.1	Integration solutions	76
8.2.2	Processes	76
8.2.3	Ports and links	77
8.2.4	Tasks and slots	79
8.2.5	Datatypes	80
8.3	Concrete syntax	81
8.4	General-purpose toolkit	81
8.5	Summary	82
9	Software Development Kit	91
9.1	Introduction	92
9.2	The framework layer	92
9.2.1	Messages	92
9.2.2	Tasks	94
9.2.3	Ports	95
9.2.4	Processes	96
9.2.5	Adapters	97
9.2.6	The Runtime System	98

9.3	The general-purpose toolkit layer	103
9.4	Summary	105
10	Model-to-text Transformations	107
10.1	Introduction	108
10.2	Transforming processes	108
10.3	Transforming ports	109
10.4	Transforming tasks	110
10.5	Transforming communicators	111
10.6	Generating the starter	112
10.7	Summary	113
11	Error Detection in Integration Solutions	115
11.1	Introduction	116
11.2	The Meta-information database	118
11.3	The Event Handler	120
11.4	The Error Detector	125
11.4.1	Finding correlations	125
11.4.2	Finding the artefacts involved in a correlation	126
11.4.3	Finding sub-correlations	128
11.4.4	Finding failing rules	129
11.4.5	Verifying correlations	132
11.5	Complexity analysis	133
11.5.1	On the implementation	134
11.5.2	Handling events	134
11.5.3	Detecting errors	135
11.6	Fault tolerance experiments	137
11.6.1	Experimentation patterns	137
11.6.2	Experimentation parameters and variables	141
11.6.3	Experimentation results	143
11.7	Summary	149
12	Case Studies	153
12.1	Introduction	154
12.2	Café	155
12.2.1	The software ecosystem	156

12.2.2	Solution	156
12.2.3	Error detection rules	157
12.2.4	Experimental results	158
12.3	Unijuí University	159
12.3.1	The software ecosystem	159
12.3.2	Solution	160
12.3.3	Error detection rules	161
12.3.4	Experimental results	162
12.4	Huelva’s County Council	163
12.4.1	The software ecosystem	164
12.4.2	Solution	164
12.4.3	Error detection rules	166
12.4.4	Experimental results	166
12.5	Travel Search	168
12.5.1	The software ecosystem	168
12.5.2	Solution	168
12.5.3	Error detection rules	169
12.5.4	Experimental results	171
12.6	Travel Booking	172
12.6.1	The software ecosystem	172
12.6.2	Solution	173
12.6.3	Error detection rules	174
12.6.4	Experimental results	174
12.7	An experiment using JBI adapters	176
12.8	Summary	177

IV Final Remarks

13	Conclusions	181
-----------	--------------------------	------------

Bibliography	187
---------------------------	------------

List of Figures

1.1	Overall picture of our contributions	11
1.2	Timeline of our contributions	12
3.1	Processing order items individually (from Hohpe and Woolf [54]) ..	35
4.1	Conceptual model of Camel	38
4.2	Café integration solution designed with Camel	43
5.1	Conceptual model of Spring Integration	46
5.2	The Café integration solution designed with Spring Integration	50
6.1	Conceptual model of Mule	54
6.2	The Café integration solution designed with Mule	58
7.1	Abstraction levels in the Model-Driven Architecture	64
7.2	Abstraction of a Software Factory	66
8.1	Conceptual map of our Domain-Specific Language	72
8.2	Typical integration solution designed with Guaraná DSL	74
8.3	Main constructors of Guaraná DSL	75
8.4	Partial view of Guaraná DSL's general-purpose toolkit	82
8.5	The Enquirer pattern	86
8.6	The Normaliser pattern	87
8.7	The Scatter-Gather pattern	88
8.8	The Claim Check pattern	88
8.9	The Message Bridge pattern	88
9.1	Packages of which our framework is composed	92

9.2	Message model	93
9.3	Task model	94
9.4	Port model	95
9.5	Process model	97
9.6	Adapter model	98
9.7	Task-based runtime model	98
9.8	Initialising the Runtime System	99
9.9	Creating and starting monitors	100
9.10	Creating and starting workers	101
9.11	Executing a <code>WorkUnit</code>	102
9.12	Task model in the toolkit	103
9.13	Adapter model in the toolkit	104
11.1	Abstract view of the monitor for error detection	116
11.2	Sample integration solutions	118
11.3	Model of the Meta-information database	119
11.4	Textual syntax for error-detection rules	119
11.5	Syntactic sugar for error-detection rules	119
11.6	Sample error-detection rules	120
11.7	Model of the Event Handler	121
11.8	Sample work graph	123
11.9	Model of Error Detector	124
11.10	Sample correlation	127
11.11	Artefacts involved in a correlation	128
11.12	Correlation that does not satisfy a rule due to excess of bindings	130
11.13	Sub-correlation that causes a rule to fail due to lack of bindings	131
11.14	The Pipeline pattern	137
11.15	The Dispatcher pattern	138
11.16	The Merger pattern	138
11.17	The Request-Reply pattern	139
11.18	The Splitter pattern	139
11.19	The Aggregator pattern	140
11.20	Experimental results for the Pipeline pattern	140
11.21	Experimental results for the Dispatcher pattern	142
11.22	Experimental results for the Merger pattern	144
11.23	Experimental results for the Request-Reply pattern	146
11.24	Experimental results for the Splitter pattern	148

11.25 Experimental results for the Aggregator pattern	150
12.1 The Café integration solution	157
12.2 Error detection rules for the Café solution	158
12.3 Experimental results for the Café solution	159
12.4 The Unijuí University integration solution	161
12.5 Error detection rules for the Unijuí University solution	162
12.6 Experimental results for the Unijuí University solution	163
12.7 The Huelva's County Council integration solution	165
12.8 Error detection rules for the Huelva's County Council solution	166
12.9 Experimental results for the Huelva's County Council solution	167
12.10 The Travel Search integration solution	169
12.11 Error detection rules for the Travel Search solution	170
12.12 Experimental results for the Travel Search solution	171
12.13 The Travel Booking integration solution	173
12.14 Error detection rules for the Travel Booking solution	174
12.15 Experimental results for the Travel Booking solution	175
12.16 Experimental results for the Café solution using JBI adapters	176

List of Tables

2.1	Maintainability measures of Camel, Spring Integration, and Mule ..	21
2.2	Maintainability measures of Guaraná	25
8.1	Concrete syntax	81
8.2	Router tasks	83
8.3	Modifier tasks	84
8.4	Transformer tasks	85
8.5	Stream dealer tasks	85
8.6	Mapper tasks	86
8.7	Communicator tasks	86
8.8	Composite tasks	87
8.9	Configuration patterns	89
11.1	Notation used in our error-detection complexity analysis	133

List of Programs

11.1	Algorithm to handle events	122
11.2	Algorithm to detect errors	124
11.3	Algorithm to find correlations	126
11.4	Algorithm to find the artefacts involved in a correlation	127
11.5	Algorithm to find sub-correlations	128
11.6	Algorithm to find failing rules	129
11.7	Algorithm to verify correlations	132

Acknowledgements

*Knowledge is in the end based on acknowledgement.
Ludwig J.J. Wittgenstein, Austrian philosopher (1889-1951)*

The first stone of big projects perpetuate dates and people. My personal project for this PhD started on the 6th of September, 2006. This day, my wife and I contacted Dr. José Miguel Toro Bonilla by sending him an e-mail message regarding working on our doctorates at his University. We did not know this professor before, so we were quite anxious about his reply. We knew that this, apparently simple, reply could change our future. Quickly José Miguel made the necessary contacts to give us a positive reply. Two days after, we got a *“We’ll be pleased to welcome you aboard!”* message from professor Dr. Rafael Corchuelo, today my supervisor. Since that day, Rafael spared no effort to help us to realise this project. Therefore, I am deeply thankful for the opportunity and help these two professors in the “Old World” gave us the opportunity to make our dreams in the “New World” come true. In a special manner, I would like to thank Rafael for his patience, support, and dedication during this learning period in Seville, Spain. I could not forget to thank my colleagues from the “TDG family” for their support and friendship.

I am also deeply thankful for the PhD scholarship granted by the Evangelischer Entwicklungsdienst e.V. (EED) and all the other support received from them, without which this project would not have been possible. Last, but not least, I would like to thank the external reviewers, namely: Dr. Vitor Manuel Basto Fernandes (Instituto Politécnico de Leiria, Portugal), Dr. Domenico Talia (Università della Calabria, Italy), Dr. Fernando Moreira (Universidade Portucalense, Portugal), Dr. Pavol Mederly (Slovak University of Technology in

Bratislava, Slovakia), Dr. Rita Francese (Università degli Studi di Salerno, Italy), Dr. Schahram Dustdar (Vienna University of Technology, Austria), Dr. Hongji Yang (De Montfort University, United Kingdom), and Dr. Maria Ganzha (University of Gdańsk, Poland).

Abstract

*The beginnings of all things are small.
Marcus T. Cicero, Roman philosopher (106 BC - 43 BC)*

Typical companies rely on their software ecosystems to support and optimise their business processes. Software ecosystems are composed of many applications that were not usually designed taking integration into account. Enterprise Application Integration provides methodologies and tools to design and implement integration solutions. The Enterprise Application Integration community has adopted the catalogue of integration patterns proposed by [Hohpe and Woolf](#) as a cookbook to design and implement integration solutions. Furthermore, there are a few software tools to help software engineers devise enterprise application integration solutions that are based on integration patterns. Some companies are interested in adapting these software tools to support their domain-specific tools to specific contexts.

In this dissertation, our research hypothesis is that the current software tools are not so easy to maintain as expected, thus increasing the costs of these adaptation process. Our goal in this dissertation is to support the thesis that it is possible to devise a domain-specific language and a set of domain-specific tools to design and implement Enterprise Application Integration solutions that are easier to maintain than the current software tools. Our core contribution consists of a Domain-Specific Language that software engineers can use to represent the models they design for their integration problems at a high-level of abstraction; a Software Development Kit that can be used to implement and run integration solutions; transformations that allow for the automatic translation of models into code; and a monitoring system that allows to detect possible errors during the execution of an integration solution. Our research results

indicate that our proposal is easier to maintain than the current tools. To evaluate and demonstrate the viability of the contributions in this dissertation, we present five case studies to which we applied our proposal. The results in this dissertation have been transferred to a spin-off and have been published as three journal papers, seven conference papers, and three workshop papers.

Resumen

*El inicio de todas las cosas es pequeño.
Marcus T. Cicero, Filósofo Romano (106 AC - 43 AC)*

Es común que las empresas necesiten sus ecosistemas software para dar soporte y optimizar sus procesos de negocio. Los ecosistemas software están compuestos por muchas aplicaciones que no han sido diseñadas teniendo en cuenta la integración. El campo de estudio conocido como Integración de Aplicaciones Empresariales proporciona metodologías y herramientas para diseñar e implementar soluciones de integración. La comunidad de Integración de Aplicaciones Empresariales ha adoptado el catálogo de patrones de integración, propuesto por [Hohpe y Woolf](#), como un estándar base para el diseño e implementación de soluciones de integración. Por desgracia, hay pocas herramientas para ayudar a los ingenieros a desarrollar soluciones de integración de aplicaciones empresariales basadas en dichos patrones. Algunas empresas tienen interés en adaptar dichas herramientas para dar soporte a sus herramientas específicas de dominio en contextos específicos.

Nuestra hipótesis de partida en esta tesis doctoral es que las herramientas actuales no son tan fáciles de mantener como sería deseado, lo que incrementa los costes involucrados en el proceso de su adaptación. Nuestro objetivo con esa tesis doctoral es probar la tesis de que es posible desarrollar un lenguaje específico de dominio y un conjunto de herramientas específicas de dominio para dar soporte al diseño e implementación de soluciones de Integración de Aplicaciones Empresariales que sean más fáciles de mantener que las herramientas actuales. Nuestra principal aportación está constituida por un Lenguaje Específico de Dominio (DSL) que permite a los ingenieros software representar con un alto nivel de abstracción sus modelos diseñados para los pro-

blemas de integración; un Kit de Desarrollo de Software (SDK) que se puede usar para implementar y ejecutar las soluciones de integración; transformaciones que permiten transformar de forma automática los modelos a código; y un sistema de monitorización que permite detectar posibles errores que ocurran durante la ejecución de una solución de integración. Nuestros resultados de investigación indican que nuestra propuesta es más fácil de mantener que las herramientas actuales. Con el objetivo de evaluar y demostrar que las contribuciones en esa tesis doctoral son viables, se presentan cinco casos de estudio a los que hemos aplicado nuestra propuesta. Los resultados presentados en esa tesis doctoral han sido transferidos a una *spin-off* y han resultado en tres publicaciones en revistas, siete en conferencias y tres en talleres.

Resumo

*O início de todas as coisas é pequeno.
Marcus T. Cícero, Filósofo Romano (106 AC - 43 AC)*

Geralmente as empresas precisam utilizar os seus ecossistemas de software para apoiar e aperfeiçoar os seus processos de negócio. Ditos ecossistemas são compostos de muitas aplicações, normalmente concebidas sem levar em conta sua possível integração. O campo de estudos conhecido como Integração de Aplicações Empresariais proporciona metodologias e ferramentas para a concepção e a implementação de soluções de integração. A comunidade de Integração de Aplicações Empresariais adotou o catálogo de padrões de integração, proposto por [Hohpe e Woolf](#), como um guia para a concepção e a implementação de soluções de integração. Porém, infelizmente, existem poucas ferramentas para ajudar os engenheiros de software no desenvolvimento de soluções de integração de aplicações empresariais, tendo como base esses padrões. Algumas empresas têm interesse em adaptar essas ferramentas com o objetivo de construir as suas próprias, focando em contextos específicos.

Nesta dissertação, nossa hipótese é a de que as ferramentas atuais, ao contrário do que delas se espera, não têm uma manutenção fácil, aumentando-se, assim, os custos envolvidos no processo de sua adaptação. Nosso objetivo com esta dissertação é provar a tese de que é possível desenvolver uma linguagem de domínio específico e um conjunto de ferramentas de domínio específico para apoiar a concepção e a implementação de soluções de Integração de Aplicações Empresariais cuja manutenção seja mais fácil do que a das ferramentas atuais. Nossa principal contribuição consiste: em uma Linguagem de Domínio Específico (DSL) que permite aos engenheiros de software representar, com um alto nível de abstração, os seus modelos para os problemas

de integração; em um Kit de Desenvolvimento de Software (SDK) que pode ser usado para implementar e executar soluções de integração; em transformações que automaticamente permitem levar os modelos a código; em um sistema de monitoramento para detectar erros que podem ocorrer durante a execução de uma solução de integração. Nossos resultados de pesquisa indicam que a nossa abordagem tem manutenção mais fácil do que a das ferramentas atuais. A fim de avaliar e de demonstrar que as contribuições nesta dissertação são viáveis, apresentamos cinco estudos de caso nos quais aplicamos a nossa proposta. Os resultados apresentados nesta dissertação foram transferidos para uma *spin-off* e resultaram em três publicações em periódicos, sete em congressos e três em seminários.

Abstrakt

Der Beginn aller Dinge ist klein.

Marcus T. Cicero, Römischer Philosoph (106 v.Chr - 43 v.Chr)

Typische Unternehmen bedürfen ihrer Software Ökosysteme bei der Unterstützung und Optimierung ihrer Geschäftsvorgänge. Software Ökosysteme bestehen aus mehreren Anwendungen, die üblicher Weise nicht darauf ausgelegt sind, Integration zu berücksichtigen. Unternehmensanwendungsintegration (UAI) stellt Methoden und Werkzeuge für Design und zur Implementierung von Integrationslösungen zur Verfügung. Die Gemeinschaft der Unternehmensanwendungsintegration nahm den von [Hohpe und Woolf](#) vorgeschlagenen Katalog von Integrationsstandards als Rezeptbuch für die Erarbeitung und Implementierung von Lösungen. Zudem stehen den Softwareingenieuren einige auf Integrationsstandards beruhende Softwarewerkzeuge für Lösungen zur Unternehmensanwendungsintegration zur Verfügung. Einige Unternehmen haben Interesse daran, diese domainspezifische Softwarewerkzeuge an spezielle Kontexte anzupassen.

In der vorliegenden Arbeit besteht unsere Hypothese darin, dass die derzeitigen Softwarewerkzeuge nicht so einfach wie erwartet aufrecht zu erhalten sind und damit die Kosten dieses Anpassungsprozesses steigern. Unser Ziel in dieser Forschungsarbeit ist es, die These zu untermauern, dass es möglich ist, eine domänenspezifische Sprache und eine Palette von domänenspezifischen Werkzeugen zu empfehlen, um Lösungen für unternehmerische Integrationsanwendungen zu entwerfen und zu implementieren, die einfacher zu unterhalten sind als die herkömmlichen Softwarewerkzeuge. Unser Hauptbeitrag besteht in einer domänenspezifischen Sprache, die Softwareingenieure benutzen können, um die Modelle darzustellen, die sie für ihre Integra-

tionsprobleme auf einem hohen Abstraktionsniveau entwerfen; ein Softwareentwicklungsbausatz, der benutzt werden kann, um Integrationslösungen zu implementieren und laufen zu lassen; Veränderungen, die automatische Umwandlung von Modellen in Codes ermöglichen; und ein Überwachungssystem, das es erlaubt, mögliche Fehler in der Anwendung einer Integrationslösung zu erfassen. Unsere Forschungsergebnisse deuten darauf hin, dass unsere Vorschläge einfacher als die herkömmlichen Werkzeuge zu unterstützen sind. Um die Machbarkeit der hier vorgestellten Beiträge zu evaluieren und zu demonstrieren, unterbreiten wir fünf Fallstudien, auf die wir unsere Vorschläge anwenden. Die Ergebnisse in der vorliegenden Doktorarbeit wurden in ein *spin-off* transferiert und wurden in drei Zeitschriften, sieben Konferenzpapers und drei Workshoppapers veröffentlicht.

Part I

Preface

Chapter 1

Introduction

*The journey of a thousand miles
begins with a single step.
Chinese proverb*

Our goal in this dissertation is to report on our work to create a set of domain-specific tools that help design, implement, and detect errors in Enterprise Application Integration solutions. In this chapter, we first introduce the context of our research work in Section §1.1. Section §1.2 presents the hypothesis that has motivated it and states our thesis. Section §1.3 introduces the collaborations we have conducted throughout the development of this dissertation. Section §1.4 summarises our main contributions. Finally, Section §1.5 describes the structure of this dissertation.

1.1 Research context

Typical companies run software ecosystems [78] that consist of many applications that support their business activities. Frequently, new business processes have to be supported by two or more applications, and the current business processes may need to be optimised, which requires interaction with other applications. However, it is common that these applications were not designed with integration concerns in mind, *i.e.*, they do not provide a programming interface. As a result, the interaction is not always a trivial task, and has to be carried out in most cases by means of the resources that belong to the applications, such as their databases, data files, messaging queues, and user interfaces.

Recurrent challenges are to make the applications inter-operate with each other to keep their data synchronised, offer new data views, or to create new functionalities [54]. In this context, many companies rely on Enterprise Service Buses to develop their integration solutions, since they provide the necessary technology to integrate disparate applications by means of exogenous workflows [15, 24]. An integration solution is deployed to the software ecosystem as a new application that provides its users with a high-level view of the integrated applications with which they can interact.

Unfortunately, applications are not usually easy to integrate due to many reasons, *e.g.*, the technologies on which they rely are different, their programming interfaces are not compatible from a semantic point of view, or they might not provide a programming interface at all, which is the case of many web applications, legacy systems, and off-the-shelf software. Additionally, integration solutions must take four important constraints into account [110], namely: first, the resources and the programming interfaces of the applications being integrated should not be modified at all since a change might seriously affect or even break other business processes; second, they must keep running independently from each other since they were designed originally without taking integration concerns into account, *i.e.*, no additional coupling must be introduced; third, the integration solution should interfere as less as possible with the normal behaviour of the integrated applications; finally, integration must be performed on demand, as new business requirements emerge and require support from Information Technology [17].

Integration solutions can be classified regarding whether they aim at the integration of functionality or data. In the former group there are two types of integration, namely: Enterprise Application Integration and Business-to-

Business Integration. The latter group includes other three types of integration, namely: Enterprise Information Integration, Extract, Transform and Load, and Mashup. In the following paragraphs we describe each type.

Enterprise Application Integration focuses on providing methodologies and tools to integrate the many heterogeneous applications of typical companies' software ecosystems. In the past few years, the application integration community has been driven by the use of patterns to solve integration problems. This has motivated a rapid growing of pattern-oriented open-source tools, such as Camel [58], Spring Integration [30], and Mule [27]. Enterprise Application Integration aims to keep a number of applications' data in synchrony or to develop new functionality on top of them, in a way that applications do not have to be changed and are possibly minimally or not affected by the integration solution [54]. From the application viewpoint, they are not aware that they take part of an integration solution.

Business-to-Business Integration is similar to the Enterprise Application Integration, except that the former does not limit the applications to belong to the same company, but aims to integrate applications from software ecosystems that belong to different companies. Although this seems a small difference, there are two major concerns, namely: first, typical applications in a company's software ecosystem are not open to the outside world; second, the infrastructure to communicate applications in different ecosystems involves using public networks in which security, reliability, of quality of service are major concerns [98].

Enterprise Information Integration allows to develop integration solutions that provide a homogeneous and on-line view of the data handled by a number of applications involved. Over this view, software engineers can execute operations using a declarative language that allows them to query and modify data in the integrated applications. An important difference between Enterprise Application Integration and Enterprise Information Integration is that there are not any data flows to keep the applications synchronised in the latter, but a global view of data. Similarly to Enterprise Application Integration, this type of integration deals with applications from the same software ecosystem.

Extract, Transform and Load is similar to the Enterprise Information Integration. The difference is that the views are off-line. Therefore, this type of integration requires a persistent database in which the data that is extracted from the integrated applications can be stored for further processing. It is common that the data that are extracted from the applications need to be transformed into a canonical schema defined by the off-line view. An off-line view is more

appropriate than an on-line view in situations in which the operations to be executed on the view require a high workload, so that executing them off-line does not affect the performance and possibly the behaviour of the original applications. In this type of integration, the persistent databases are usually referred to as Warehouses [89] or Data Marts [7], depending on whether they store company-wide data or focus only on a department, customers, or sales, to mention a few examples.

Mashups became very popular in the last years and aim to develop new applications by composing web services [72]. Mashups differ from the previous types of integration in that they can only integrate services that provide a programming interface, *i.e.*, they have to provide a programming interface that allows to interact with the service [72]. The original goal of Mashups was to provide end-users with ability to develop new views that integrate a set of web services. This made Mashups very popular and several companies released their technologies for Mashups, such as Yahoo Pipes [113], IBM Mashup Centre [57], WSO2 Mashup Server [112], SAP Rooftop [56], and Jackbe Presto [60].

In this dissertation we focus on Enterprise Application Integration from a Model-Driven Engineering perspective. Model-Driven Engineering is the driving force behind many current Software Engineering proposals that attempt to solve complex problems at a high-level of abstraction. For this reason, models are promoted as first-class citizens in every phase of the software development process; automatic code generation allows to transform these models into executable code.

1.2 Research rationale

In this section we present the hypothesis that has motivated our research work in the context of Enterprise Application Integration, and state our thesis, which we prove in the rest of the dissertation.

1.2.1 Hypothesis

Integration is expensive, but necessary as new applications sprout out. Usually, most of the functionalities and information involved in maintaining an existing business process or creating a new one can be found within

a company's software ecosystem. The reuse of these resources within the ecosystem contributes to reducing software development costs and deployment time [5, 6, 68, 88].

Unfortunately, according to Weiss [111], the cost of integration is usually 5–20 times the cost of developing new functionalities. Development companies incur these high costs when they face their work using general-purpose languages and tools like Java and its accompanying workbenches, instead of using languages and tools that are specifically tailored towards solving integration problems. Software Engineers are responsible for devising these languages and tools. Domain-specific languages are intended to provide constructs by means of which a problem can be described at an abstraction level that is close to the conceptual level; later, the models expressed using these languages can be transformed automatically into low-level technologies.

Proposals like Camel, Spring Integration, and Mule provide domain specific languages for Enterprise Application Integration. Unfortunately, they do not focus on a specific context within this domain, e.g., e-commerce, health systems, financial systems, or insurance systems, to mention a few. Each such context requires constructs to deal with standards and recommendations like RosettaNet [96], HL7 [52], SWIFT [106], and HIPAA [51]. As a conclusion, if a provider of integration solutions wishes to specialise in a specific context, it makes sense that they try to refine Camel, Spring Integration, and Mule to deal with the previous standards and recommendations. We focus on open-source proposals because they are currently very used in the market. Unfortunately, they do not seem so easy to maintain.

As a conclusion, we formulate the following hypothesis:

Companies rely on software ecosystems to support their business processes. There is an increasing need to integrate disparate resources within these software ecosystems. Companies that provide Enterprise Application Integration solutions need domain-specific tools that are easy to maintain in order to develop their customised tools for integration.

1.2.2 Thesis

The Model-Driven Engineering discipline [100] heralds the idea that constructing software building on high-level models helps reduce development

and maintenance costs [108].

Common open-source tools were not designed at a high-level of abstraction, but they were more sort of grown from source code contributed by a variety of programmers world-wide. This has resulted in poor design that seems to be difficult to maintain.

As a conclusion, we formulate the following thesis:

It is possible to devise a domain-specific language and a set of domain-specific tools to develop Enterprise Application Integration solutions within the context of Model-Driven Engineering with better maintenance measures than the state-of-the-art proposals in this field.

1.3 Collaborations

Throughout the development of this dissertation, several collaborations were conducted. Not only allowed these collaborations to gather feedback about our research results, but they also resulted in joint publications, knowledge transfer to the industry, and lots of interesting discussions, as well. In the following, we provide additional information about each collaboration.

- Newcastle University (United Kingdom): A research visit was paid to The Distributed Systems Research Group of the School of Computing Science from the 9th of February until the 3rd of April, 2009. The goal was to research on fault tolerance applied to Enterprise Application Integration. This collaboration resulted in the following publications: [21, 37–39, 41–43].
- University of Leicester (United Kingdom): A research visit was paid to the Department of Computer Science of the School of Mathematics & Computer Science from the 4th of October until the 9th of December, 2009. The focus was on presenting our research results, gathering feedback from a group of researchers working on Domain-Specific Languages, and delving into other aspects of our work.
- Polytechnic Institute of Leiria (Portugal): A research visit was paid to the Computer Science and Communication Research Centre from the 10th of October until the 16th of November, 2011. The focus was on presenting

our research results, researching on how our work can be used to support the design and implementation of integration solutions in the context of the integration problems tackled by the hosting group, applying the techniques for optimisation used by the hosting group to Guaraná, gathering feedback from a group of researchers working on Enterprise Application Integration, and delving into other aspects of our work.

- Federal University of Rio Grande do Sul (Brazil): Two workshops were held to present research results of both the Brazilian and Spanish research groups. The first workshop was held in Brazil and the second in Spain. The work carried out during this collaboration, allowed us to prepare a proposal for a joint research project. Furthermore, a student from the Federal University of Rio Grande do Sul has started her master's studies, in which she is currently researching on a novel Runtime System that can support the execution of our proposal on Java tuple spaces. Preliminary results are presented in [104].
- Intelligent Dialogue Systems, S.L. (Spain): This company works on the development of interactive virtual assistants. We transferred partial results on our Domain-Specific Language to them, so that their software engineers could try it to design models for an integration problem at their business domain. This work allowed them to visualise their integration solution at a higher-level of abstraction, instead of at code level as they were used to. Furthermore, it allowed us to have feedback that helped to improve our Domain-Specific Language. Our results were published in [22, 36].
- Intelligent Integration Factory, S.L. (Spain): This company provides services, including the design and implementation of Enterprise Application Integration solutions. We have chosen this company as a bridge that allows us to validate and transfer the research results from this doctoral thesis to the industry. Not only allows us this collaboration to contribute to the local development, but also gives us more feedback that can be used to drive future work.
- Huelva's County Council (Spain): The collaboration with this public administration was conducted at their IT Department. The goal was to use our Domain-Specific Language and Software Development Kit to solve the integration problem introduced in Section §12.4. This work contributed to validate our research results, and was documented in [1].

Our results would not have been possible without the collaboration of the following students and professors:

- Slovak University of Technology in Bratislava (Slovakia): Bradác [11] applied the constraint programming-based method described in [75, 76] to find and automatically generate optimal integration solutions. In this work, our Domain-Specific Language and Software Development Kit were studied and compared to other open source tools.
- University of Seville (Spain): Regalado [93] constructed a workbench that was based on Microsoft DSL Tools [20]. It built on an earlier version of our Domain-Specific Language. This was the first version of a graphical editor for our language.
- University of Seville (Spain): Sleiman [103] took the models serialised by the workbench constructed by Regalado [93] and proposed a runtime to execute them that was based on Microsoft Windows Workflow Foundation [13].
- University of Seville (Spain): Lobato [70] developed a new version of the workbench that focuses on the Domain-Specific Language presented in this dissertation. Furthermore, this version includes model-to-text transformations to enable automatic code generation based on the Software Development Kit presented in this dissertation, as well.
- University of Seville (Spain): We carried out a collaboration with Reina-Quintero in [43] to develop an Eclipse plug-in that supports the design and implementation of integration solutions using the Domain-Specific Language and the Software Development Kit presented in this dissertation.

1.4 Summary of contributions

To prove our thesis, we have developed Guaraná. Figure §1.1 presents an overall picture. The metamodel of Guaraná supports a number of concepts that software engineers can use to devise their integration solutions. Note that the metamodel is divided into two parts, namely: a core, which supports a subset of concepts that are assumed to be useful across a wide range of integration solutions, and a series of task toolkits, which supports subsets of tasks that are assumed to be specific to a given domain of integration.

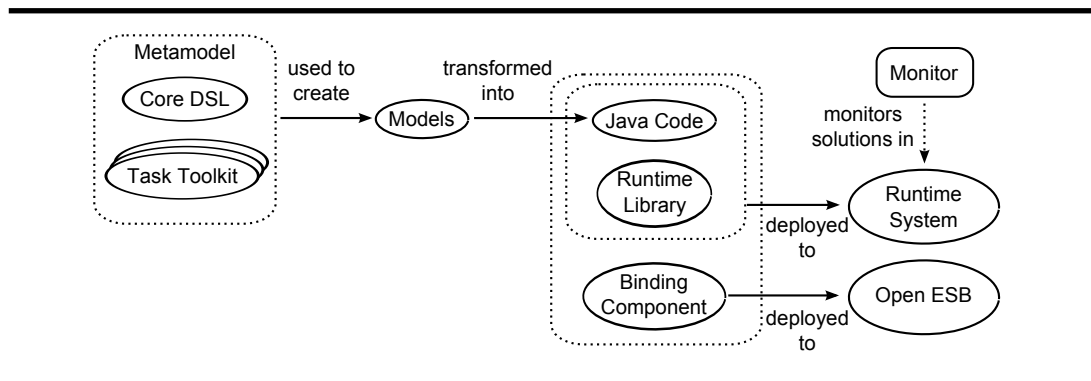


Figure 1.1: *Overall picture of our contributions.*

A software engineer can use the concepts defined in the metamodel to create his or her own models, which are specific solutions to specific integration problems. Such models are graphical and allow to devise integration solutions at a high-level of abstraction. The transformations allow to translate Guaraná models into Java code that relies on a runtime library that provides base classes to implement the concepts supported by the metamodel.

Note that the Java code plus the runtime library are not enough to implement an integration solution; it is also necessary a number of binding components. Binding components implement the low-level transport protocol necessary to interact with the applications being integrated. They are part of the Java Business Integration specification, which drives the architecture of several Enterprise Service Buses, such as the Open ESB [17, 87]. Whereas the Java code must be compiled and deployed to the Runtime System provided by Guaraná, the binding components must be configured and deployed to Open ESB independently. Integration solutions deployed to the Runtime System are monitored, to allow for the detection of possible errors.

Thus, our contributions are summarised as follows, *cf.* Figure §1.2:

- First, we have developed a Domain-Specific Language to design integration solutions. This language allows software engineers to design integration solutions to their problems at a high abstraction level, which relieves them from the burden of dealing with the rather low-level constructs provided by programming languages. Furthermore, the language supports the well-known integration patterns largely used by the integration community [54], and sets a common and yet sim-

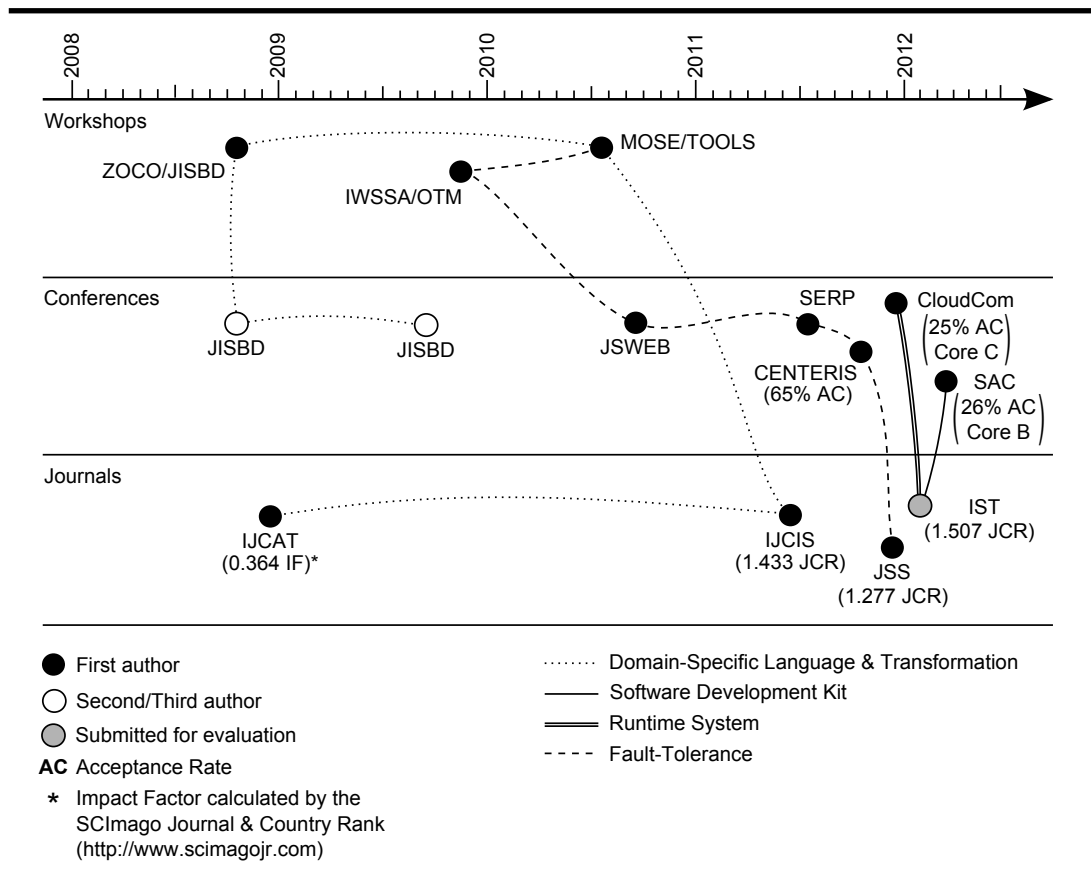


Figure 1.2: *Timeline of our contributions.*

ple vocabulary for the communication in this field, as well. In the context of this contribution, we had the following publications accepted: [22, 33, 36, 42, 43, 102].

- Second, we have developed a Software Development Kit that supports the implementation of models designed with our Domain-Specific Language. Its key feature is that its maintenance measures indicate that it should be much easier to maintain than other proposals. In the context of this contribution, we had the following publication accepted: [34]. An extended version of our work on the Software Development Kit was also submitted for evaluation to the Information and Software Technology (IST) journal.
- Third, we have developed a Runtime System to execute integration solutions implemented using our Software Development Kit. In the context

of this contribution, we had the following publication accepted: [35]. We have also extended our work on the Runtime System and included it in the article in which we describe our Software Development Kit.

- Fourth, an Eclipse-based workbench was developed to help design integration solutions using our Domain-Specific Language by means of a graphical editor. Furthermore, model-to-text transformations were attached to this workbench, thus providing software engineers with automatic code generation scripts to transform their models into executable integration solutions that rely on our Software Development Kit and are executed on our Runtime System. In the context of this contribution, we had the following publication accepted: [43].
- Fifth, we have developed a monitoring system to provide integration solutions with error monitoring. This system monitors integration solutions running on our Runtime System. We deal with a subset of faults, including failure to read from or write to a resource, missing messages, structural, and deadline errors. In the context of this contribution, we had the following publication accepted: [37–42].

1.5 Structure of this dissertation

This dissertation is organised as follows:

Part I: Preface. Comprises this introduction and Chapter §2, in which we motivate our research work and conclude that current software tools for Enterprise Application Integration do not seem as easy to maintain as they are expected to.

Part II: Background Information. Provides information about the software tools and technologies that are related to our research context. In Chapter §3, we introduce the well-known enterprise application integration patterns, which remain in the bases of our research work. In Chapter §4, we describe Camel, which is the software tool provided by the Apache Software Foundation. In Chapter §5, we describe Spring Integration, which is the software tool provided by VMware Inc. In Chapter §6, we describe Mule, which is the software tool provided by MuleSoft Inc. In Chapter §7, we give an introduction to Model-Driven Architecture and Software Factories, which are two technological approaches within Model-Driven Engineering.

Part III: Our Approach. Reports on the core contributions we made with this dissertation. This part starts at Chapter §8, with a report on the Domain-Specific Language we have developed to design integration solutions. In Chapter §9, we report on a Software Development Kit, which can be used to implement and run integration solutions. In Chapter §10, we introduce the transformations that can be used to automate the process of code generation. In Chapter §11, we report on our proposal to monitor and detect errors in integration solutions. In Chapter §12, we present five case studies in which we have used our Domain-Specific Language, Software Development Kit, transformations, and the fault tolerance proposal to demonstrate their viability.

Part IV: Final Remarks. Concludes this dissertation and highlights a few future research directions in Chapter §13.

Chapter 2

Motivation

*The great tragedy of Science: the slaying of
a beautiful hypothesis by an ugly fact.*

Thomas H. Huxley, English biologist (1825-1895)



Although it is possible to use current tools to design and implement Enterprise Application Integration solutions, it is still necessary to provide domain-specific tools that are easy to maintain in order to customise them for a specific context. Our goal in this chapter is to introduce a set of measures that can be used as an indicator of how maintainable a tool is, present the values calculated for each measure, and to motivate the need for tools with a good architecture that facilitates customising them for specific contexts. It is organised as follows: in Section §2.1, we introduce the chapter; in Section §2.2, we describe the fifteen most usual maintainability measures; in Section §2.3, we present the values calculated for each measure in the tools we have analysed; in Section §2.4, we discuss on the reasons why we think that current tools are not as maintainable as they should, and report on the results of the maintainability measures regarding our proposal; finally, Section §2.6 summarises the main ideas in this chapter.

2.1 Introduction

Companies rely on applications to support their business activities. Frequently, these applications are legacy systems, packages purchased from third parties, or developed at home to solve a particular problem. This usually results in heterogeneous software ecosystems, which are composed of applications that were not usually designed taking integration into account. Integration is necessary, chiefly because it allows to reuse two or more applications to support new business processes, or because the current business processes have to be optimised by interacting with other applications within the software ecosystem. Enterprise Application Integration provides methodologies and tools to design and implement integration solutions. The goal of an Enterprise Application Integration solution is to keep a number of applications' data in synchrony or to develop new functionality on top of them, so that applications do not have to be changed and are not disturbed by the integration solution [54].

In the last years, several tools have emerged to support the design and implementation of integration solutions. Hohpe and Woolf [54] documented many patterns found in the development of integration solutions. Camel, Spring Integration, and Mule are the most popular open-source tools that provide support for some of these integration patterns. Camel provides a fluent API [31] that software engineers can use programmatically or by means of a graphical editor. In both cases, the integration solution is implemented using a Java, Scala, or XML Spring-based configuration files. Spring Integration was built on top of the Spring Framework container, and provides a command-query API [31]. This tool can be used programmatically or by means of a graphical editor. Integration solutions are implemented using either Java code or an XML Spring-based configuration file. The architecture of Mule got inspiration from the concept of enterprise service bus. Software engineers count on a command-query API [31] to use this tool programmatically, or a workbench to design and implement integration solutions using a graphical editor. Integration solutions are implemented using either Java code or an XML Spring-based configuration file. In earlier versions, Mule supported a limited range of integration patterns; version 3.0 resulted in a complete re-design whose focus was on supporting the majority of integration patterns. As of the time of writing this dissertation, Camel, Spring Integration, and Mule are at version 2.7.1, 2.0.3, and 3.1, respectively. In the rest of the dissertation, we implicitly refer to these versions.

The Model-Driven Engineering discipline promotes models as first-class

citizens in every phase of the software development process [100]. Counting on this discipline, software engineers can devise solutions to their problems at a proper abstraction level, without having to deal with the rather low-level constructs provided by programming languages, and the process of code generation from models can be automated [9]. In recent versions, Camel, Spring Integration, and Mule have included support to raise the level of abstraction by means of graphical domain-specific languages that can be used in their visual editors, and they have introduced a number of mechanisms to automate code generation. However, as we discuss later in this chapter, Camel, Spring Integration, and Mule do not seem to be so easy to maintain. Consequently, it is still necessary to research on devising easier-to-maintain tools to support the design and implementation of integration solutions, which is our purpose in this dissertation.

2.2 Problems

Although it is commonly agreed that maintaining software means changing it, there does not seem to be a consensus on the different types of maintenance. In this dissertation, we use the classification proposed by IEEE [59], according to which maintenance can be corrective, perfective, or adaptive. Corrective maintenance aims to repair a software system to eliminate faults that might cause the system to deviate from its normal processing. Perfective maintenance aims to modify a software system to improve the performance of its current functionalities or even to improve its maintainability. Adaptive maintenance focuses on adapting a software system to make it usable in a new execution environment or business context.

In this dissertation, we are interested in adaptive maintenance, which is very important for companies that need to build their own tools building on existing tools. Many companies rely on open-source tools that can be adapted to a specific context within their business domain. For example, a company that develops Enterprise Application Integration solutions may need tools that focus on specific contexts such as e-commerce, health systems, financial systems, and insurance systems to meet standards and recommendations like RosettaNet [96], HL7 [52], SWIFT [106], and HIPPA [51], respectively.

It is not new that how a software system was designed and implemented, has an impact on its maintenance costs [8, 28, 61, 101]. In both design and implementation, software engineers need to pay attention to readability, un-

derstandability, and complexity. The resulting models and source code must be easy to read and understand, because it is very common that the people who work on them shall not maintain them. The complexity of the algorithms should be kept low, not only for performance reasons, but because it makes it easier for a software engineer to follow their execution flows and debug them. Thus, to reduce the costs involved in the adaptation of a software system to a specific context, it is very important that the software system was designed taking into account issues that have a negative impact on maintenance.

How costly a software tool is to be maintained depends on a variety of properties. The Software Engineering community uses a number of measures proposed by Chidamber and Kemerer [16], Henderson-Sellers [50], Martin [73], and McCabe [74] to have an overall idea of how difficult maintaining a system can be. Below, we describe the fifteen most usual such measures:

NOP: Number of packages that contain at least one class or interface. This measure can be used as an indicator of how much effort it is required to understand how packages are organised; note that this provides the overall picture of the design of a system [26]. The greater this value, the more effort shall be required.

NOC: Number of classes. This and the following measure (NOI) can be used as indicators of how much effort shall be required to understand the source code of a software system. The greater this value, the more difficult it is to understand a software system.

NOI: Number of interfaces.

NOH: Number of immediate children classes of a class. This measure can be used as an indicator of the potential impact that a class may have in a software system if it is modified [16]. The greater this value, the greater the chances that the abstraction defined by the parent class is poorly designed.

LOC: Number of lines of code, excluding blank lines and comments. In general, the greater this value, the more effort shall be required to maintain a software system.

NOM: Number of methods in classes and interfaces. This measure can be used as an indicator for the potential reuse of a class. According to Lorenz and Kidd [71], and Chidamber and Kemerer [16], a large number of methods may indicate that a class is likely to be application specific, limiting the possibility of reuse.

- NPM:** Number of parameters per method. This measure can be used as an indicator of how complex it is to understand and use a method. According to Henderson-Sellers [50], the number of parameters should not exceed five. If it does, the author suggest that a new type must be designed to wrap the parameters into a unique object. The greater this value, the more difficult it is to understand a method.
- WMC:** Weighted sum of the McCabe cyclomatic complexity [74] for all methods in a class. This measure can be used as an indicator of how difficult understanding and then modifying the methods of a class shall be [16]. The greater this value, the more effort is expected to maintain a class.
- DBM:** Depth of nested blocks in a method. This measure can be used as an indicator of how expensive debugging a piece of code can be. According to Henderson-Sellers [50], this value should not exceed five. If it does, the author suggests that the method should be broken into other methods. The greater this value, the more complex an algorithm is.
- LCM:** Lack of cohesion of methods. In this context, cohesion refers to the number of methods that share common attributes in a class. It is calculated with the Henderson-Sellers LCOM* method [50]. A low value indicates a cohesive class; contrarily, a value close to one indicates lack of cohesion and suggests that the class might better be split into two or more subclasses because there can be methods that should probably not belong to that class.
- MLC:** Number of lines in methods, excluding blank lines and comments. According to Henderson-Sellers [50], this value should not exceed fifty. If it does, the author suggests to split this method into other methods to improve readability and maintainability. The greater this value, the more difficult it is to understand and maintain a method.
- AFC:** Afferent coupling. This measure is defined as the number of classes outside a package that depend on one or more classes inside that package. The greater this value, the more complex maintenance becomes because there are more dependencies between classes [73, 83, 115]. Furthermore, larger values of afferent coupling can be used as an indicator that the package is critical for the software system and then maintenance in this package must be performed carefully not to introduce problems in the dependent classes.
- EFC:** Efferent coupling. This measure is defined as the number of classes inside a package that depend on one or more classes outside the package.

The greater this value, the more likely that maintenance shall have an impact on a package [73, 83, 115].

ABS: Degree of abstractness of a software system. This measure can be used as an indicator of how customisable a software system is [73]. The greater this value, the easier to customise the software system.

MCC: The McCabe cyclomatic complexity. This measure can be used as an indicator of how complex the algorithm in a method is. According to [74], this value should not exceed ten. The greater this value, the more difficult it is to maintain a piece of code.

2.3 Analysis of current solutions

We have calculated the maintainability measures regarding the core implementation of Camel, Spring Integration, and Mule, *i.e.*, we do not take into account the code required to implement the adapters that are required to interact with the applications being integrated. We do not consider this code because it is peripheral and, more often than not, comes from other open-source projects that are maintained separately. Table §2.1 summarises the results.

The architecture of the tools we have analysed is organised into several packages: 54 in Camel, 32 in Spring Integration, and 124 in Mule. Although Mule has more than double as many packages as Camel, they have approximately the same total number of classes in their packages. Nevertheless, there are cases in which the maximum number of classes in a package reaches 96 in Camel, 58 in Spring Integration, and 51 in Mule. These values show that Camel has almost double as many classes in a package as Spring Integration or Mule. The same happens regarding the number of interfaces. Consequently, Camel has the highest standard deviation and mean values per package regarding both, classes and interfaces, which has an impact on the understandability of its packages. Spring Integration is the only tool that has a low value for the standard deviation regarding the number of interfaces.

The maximum number of immediate children classes of a class also varies very much: 69 in Camel, 11 in Spring Integration, and 28 in Mule. If considered the mean and the standard deviation values per class, Camel has the highest values, which indicates that the abstraction defined by parent classes tend to be poorly designed. Other values that are impressive for these tools are regarding the total number of lines of code, which is very high in every

Measure	Camel				Spring Integration				Mule			
	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max
NOP	54	-	-	-	32	-	-	-	124	-	-	-
NOC	730	13.52	19.55	96	269	8.41	10.52	58	733	5.91	7.40	51
NOI	140	2.59	9.07	58	40	1.25	1.84	9	209	1.69	3.28	18
NOH	493	0.68	3.77	69	147	0.55	1.54	11	337	0.46	1.82	28
LOC	62,439	-	-	-	14,929	-	-	-	67,090	-	-	-
NOM	7,015	9.61	15.36	192	1,431	5.32	5.60	39	5,158	7.04	10.23	129
NPM	-	0.93	1.05	11	-	1.13	0.94	9	-	0.92	1.07	19
WMC	12,903	17.68	27.37	346	2,628	9.77	11.27	68	10,537	14.38	21.92	262
DBM	-	1.37	0.79	8	-	1.44	0.86	6	-	1.43	0.87	8
LCM	-	0.29	0.35	1	-	0.22	0.33	0.94	-	0.23	0.34	1.33
MLC	34,839	4.52	8.15	141	8,264	5.65	9.59	110	35,989	6.16	10.99	180
AFC	-	30.63	89.34	542	-	12.69	26.65	146	-	22.90	56.25	493
EFC	-	12.54	17.83	87	-	8.44	9.84	55	-	6.22	6.76	38
ABS	-	0.15	0.21	1	-	0.27	0.25	1	-	0.33	0.33	1
MCC	-	1.67	2.06	46	-	1.80	2.04	30	-	1.80	2.01	33

NOP = Number of packages; NOC = Number of classes; NOI = Number of interfaces; NOH = Number of immediate children classes of a class; LOC = Number of lines of code; NOM = Number of methods in classes and interfaces; NPM = Number of parameters per method; WMC = Weighted sum of the McCabe cyclomatic complexity for all methods in a class; DBM = Depth of nested blocks in a method; LCM = Lack of cohesion of methods; MLC = Number of lines in methods, excluding blank lines and comments; AFC = Afferent coupling; EFC = Efferent coupling; ABS = Degree of abstractness of a software system; MCC = The McCabe cyclomatic complexity.

Table 2.1: Maintainability measures of Camel, Spring Integration, and Mule.

tool, chiefly for Camel and Mule. These tools are 62,439 and 67,090 lines of code respectively, contrarily to 14,929 in Spring Integration.

When analysing the methods in classes and interfaces, we found that Camel has 7,015 methods compared to the 1,431 and the 5,158 found for Spring Integration and Mule, respectively. Most probably, the difference amongst Spring Integration and the other tools is because it has less than a half the number of classes and interfaces of Camel and Mule. The values that stand out are the maximum number of methods per class/interface calculated in Camel and Mule, which are 192 and 129 respectively, contrarily to 39 in Spring Integration. If we look at the maximum number of parameter per method, it is also impressive how large it is, chiefly in Camel and Mule: 11 and 19 respectively. Spring Integration has a maximum of 9 parameters. These values indicate that some classes in these tools are likely too application specific, with a limited possibility to be reused; furthermore, this makes some of their

methods difficult to understand, chiefly in the case of Camel and Mule.

The weighted method complexity calculated also demonstrates a high cyclomatic complexity within classes, chiefly for Camel and Mule. In these tools, the total weighted method complexity was 12,903 and 10,537, respectively. For Spring Integration, the cyclomatic complexity is 2,628, which is not so high when compared to Camel and Mule. Nevertheless, not only the total cyclomatic complexity is high, but also the mean, the standard deviation, and the maximum. Camel, Spring Integration, and Mule have maximum values of 346, 68, and 262, respectively. The depth of nested blocks in a method is similar in every tool. If we consider the mean and maximum values, Camel has 1.37 and 8, Spring Integration has 1.44 and 6, and Mule has 1.43 and 8, respectively. Similarly, the mean and the maximum values for the lack of cohesion of methods is similar in every tool. Camel has 0.29 and 0.35, Spring Integration has 0.22 and 0.33, and Mule has 0.23 and 0.34. Counting the number of lines of code inside methods, we found Camel has a total number of 34,839, Spring Integration has 8,264, and Mule has 35,989, which if compared to the total number of lines of code, represents 0.55%, 0.55%, and 0.53% of these values, respectively. It means there are many attributes declared in classes. The maximum value calculated demonstrate that there are some methods with until 141 lines of code in Camel, 110 in Spring Integration, and 180 in Mule. These values indicate more effort might be necessary to maintain and understand the methods in these tools.

Regarding the coupling of classes, the values for the afferent and efferent coupling in every tool are very high. Camel has the highest value for the afferent coupling, followed by Mule and then Spring Integration, with a mean of 30.63, 12.69, and 22.90, respectively. It is also very impressive the standard deviation, chiefly for Camel and Mule, which have 89.34 and 56.25, respectively. The maximum values are also very high, being 542 for Camel, 146 for Spring Integration, and 493 for Mule. These values suggest that much attention must be paid when performing maintenance in the classes of a package. The mean for the efferent coupling varies from 12.54 in Camel and 8.44 in Spring Integration, to 6.22 in Mule. The maximum values are not so impressive as for the afferent coupling, but they are still very high. In Camel, the maximum efferent coupling is 87; in Spring Integration, it is 55; in Mule it is 38. These figures suggest that the classes inside a package have a large number of dependencies on outside classes and maintenance has to be done carefully; as a conclusion, the impact on maintenance should not be neglected at all.

The values for the degree of abstractness indicates that Camel is the less

abstract tool. The mean value for Camel is 0.15, followed by 0.27 for Spring Integration, and 0.33 for Mule. The results indicate that these tools are not so easy to customise, chiefly Camel because its mean value is very low. The values calculated for the McCabe cyclomatic complexity indicate that there are cases in which they are extremely high. This is indicated by the maximum values, which reach 46, 30, and 33 in Camel, Spring Integration, and Mule, respectively. Consequently, they are also very complex tools, which may have a serious impact on their maintenance.

From the analysis of the fifteen maintainability measures, it follows that the tools we have analysed may have problems regarding maintenance, chiefly adaptive maintenance, which is our main concern in this dissertation.

2.4 Discussion

In the literature, there are several references to assessing the maintainability of open-source tools [64, 97, 99, 114, 116–118]. There is a great evidence that open-source tools need to improve their maintainability. Below, we report on a few issues that have an impact on the maintainability of open-source tools:

- The majority of the open-source tools are developed by a heterogeneous community of software engineers, who share common interests.
- Current software engineers involved in a project and new software engineers that join that project, mostly have only the source code and the archives of the project's mailing list as a source of information. Furthermore, it is not so easy to read this information, find, and understand the relationships amongst the concepts involved in the architecture of a software system [23]. The idea of contributing with an open-source project is much more associated with writing code than with writing documentation. The lack of documentation may lead to different interpretations of the same architecture, thus degrading the quality of new contributions and consequently having a deleterious impact on maintainability.
- Open-source development is usually code-centric instead of model-centric. By models we mean a graphical representation of the architecture using well-known modelling languages, such as the Unified Modelling Language; by code, we mean the source code written using a general-purpose programming language, such as Java. Sometimes it is

possible to find sketches with informal descriptions of the software system or part of its architecture. However, they do not frequently reflect the current state of the software system and its actual architecture. They are intended to give an overview of the software system, its functionalities, or to describe part of the architecture to end-users, not to software engineers that contribute to the open-source tool. Thus, they are not useful and reliable to describe the architecture of a software system, which is essential for a software engineer that aims to maintain a tool.

- Although in most open-source projects there is a person or a group of people who are responsible for managing the contributions made by the community involved in the project, not every contribution is checked. Furthermore, along the life-cycle of a software tool, the same piece of code may have been modified by several software engineers with different levels of expertise. This may lead to situations in which code with quality below the expected standards is introduced in the source code of a tool [99]. This kind of contributions certainly may have an impact on readability, understandability, and the complexity of the architecture.
- The evolution of open-source tools is driven by community requirements [114]. There are requirements that can cause an important impact on the architecture of a tool, which makes it difficult to embrace them in a short time. The majority of open-source projects rely on software engineers that contribute to them during their leisure time rather than work time [114]. Thus, it may take more time to discuss and find the best solution to a problem in a short time. Consequently, it is not uncommon that modifications or even workarounds are done to temporarily accommodate the new requirements, leaving for the future main releases the proper and correct maintenance in the architecture.

As a conclusion, it is not surprising that the maintenance measures of Camel, Spring Integration, and Mule are so disappointing. This motivates the need for researching on how to develop a software tool with a better design, which can provide better values for each maintainability measure.

2.5 Our proposal

We have developed a new software tool to support the implementation of Enterprise Application Integration solutions. We refer to this tool as Guaraná.

Measure	Guaraná				Deltas with respect to other software tools			
	Total	Mean	Dev.	Max	Total	Mean	Dev.	Max
NOP	18	-	-	-	-52	-	-	-
NOC	79	4.39	3.09	11	-498	-4.89	-9.39	-57.33
NOI	9	0.50	0.76	2	-121	-1.34	-3.97	-26.33
NOH	59	0.75	2.05	10	-267	0.19	-0.33	-26
LOC	2,878	-	-	-	-45,275	-	-	-
NOM	369	4.67	4.61	24	-4,166	-2.65	-5.79	-96
NPM	-	1.20	1.04	4	-	0.20	0.02	-9
WMC	498	6.30	6.30	37	-8,191	-7.64	-13.88	-188.33
DBM	-	1.24	0.74	4	-	-0.18	-0.10	-3.33
LCM	-	0.14	0.27	0.91	-	-0.11	-0.07	-0.19
MLC	1,748	4.72	6.43	54	-24,616	-0.72	-3.14	-89.67
AFC	-	6.94	14.33	47	-	-15.13	-43.08	-346.67
EFC	-	4.17	2.81	11	-	-4.90	-8.66	-49
ABS	-	0.54	0.35	1	-	0.29	0.08	0
MCC	-	1.35	0.91	8	-	-0.41	-1.13	-28.33

NOP = Number of packages; NOC = Number of classes; NOI = Number of interfaces; NOH = Number of immediate children classes of a class; LOC = Number of lines of code; NOM = Number of methods in classes and interfaces; NPM = Number of parameters per method; WMC = Weighted sum of the McCabe cyclomatic complexity for all methods in a class; DBM = Depth of nested blocks in a method; LCM = Lack of cohesion of methods; MLC = Number of lines in methods, excluding blank lines and comments; AFC = Affferent coupling; EFC = Efferent coupling; ABS = Degree of abstractness of a software system; MCC = The McCabe cyclomatic complexity.

Table 2.2: *Maintainability measures of Guaraná.*

As of the time of writing this dissertation, Guaraná is at version 1.3.0. Its design provides better values for maintainability measures, which suggests that our proposal is more maintainable and thus easier to adapt to a specific context than Camel, Spring Integration, or Mule. Table §2.2 summarises the results.

The architecture of Guaraná is organised into 18 packages, and the maximum number of classes in a package is no more than 11. Furthermore, Guaraná provides no more than 9 interfaces in these packages. The standard deviation calculated for the number of classes and interfaces per package is very low, 3.09 and 0.76, respectively. Note that, for the total number of packages, the delta indicates that Guaraná has 52 packages less than the other software tools. The difference respect to the other software tools is even greater when considering the deltas for the number of classes and for the number of interfaces, which are -498 and -121 , respectively. These values indicate that maintenance in Guaraná is not expected to be difficult. The maximum number

of immediate children classes of a class is no more than 10, with a mean of 0.75 per package. These values situate Guaraná with a delta of -267 respect to the other software tools. These values indicate that the abstraction defined by the parent class is well designed in Guaraná. The implementation of Guaraná has a total number of 2,878 lines of code, which correspond to 45,275 less lines of code than the other software tools.

We have analysed the methods in classes/interfaces and found that Guaraná has in total 369 methods, with a maximum number of 24 methods per class/interface, and no more than 4 parameters per method. When comparing Guaraná to the other software tools, the deltas reveal that Guaraná has 4,166 less methods, and if we look at the maximum number of parameter per method it has 9 less parameters. These values indicate that classes in Guaraná are expected to be more reusable and its methods not so difficult to understand. The weighted method complexity calculated indicates a low cyclomatic complexity within classes of Guaraná. The total weighted method complexity is 498, the mean and the standard deviation were 6.30, and the maximum is 37. The delta for the other software tools reveal that Guaraná has 8,191 less total weighted method complexity and a standard deviation of 13.88 less as well. The depth of nested blocks in a method in Guaraná has a mean of 1.24, a standard deviation of 0.74, and a maximum of 4. If we consider the maximum value in Guaraná, it is 3.33 less than in other software tools. Regarding the lack of cohesion of methods, values are very low in Guaraná. It presents a mean of only 0.14, which reveals that the other software tools have a delta of -0.11 in average with respect to Guaraná. Counting the number of lines of code inside methods, we found Guaraná has a total number of 1,748, which, if compared to the total number of lines of code in Guaraná, represents 0.61% of this value. Furthermore, there is no method with more than 54 lines of code, being the average 4.72 lines of code per method. The other software tools have 24,616 more method lines of code, and if we consider the maximum number of lines of code per method, they have 89,67 more lines than in Guaraná. These values indicate that classes are expected to be easier to understand and maintain.

Regarding the coupling of classes, the values for the afferent and efferent coupling in Guaraná are not very high. The afferent coupling has values 6.94, 14.33, and 47 as mean, standard deviation, and maximum, respectively. The efferent coupling has values 4.17, 2.81, and 11 as mean, standard deviation, and maximum, respectively. The average afferent and efferent coupling in Guaraná are 15.13 and 4.90 less than in other software tools, respectively. These values suggest that the classes in Guaraná do not have a high number of dependencies and maintenance is expected to be easy. The values for the de-

gree of abstractness, indicate Guaraná has a mean value of 0.54, which situates Guaraná is 0,29 in average more abstract than the other software tools. These values suggest that it shall not be complicated to customise Guaraná. The values calculated for the McCabe cyclomatic complexity have indicated that the maximum value is 8, which situates Guaraná with 28.33 less complexity than other software tools. These values indicate the architecture in Guaraná is well designed and maintenance is expected to be easy.

2.6 Summary

In this chapter, we have reported on the fifteen most usual measures that can be used as an indicator of how maintainable a software tool is. We have calculated these measures for Camel 2.7.1, Spring Integration 2.0.3, and Mule 3.1. The resulting values indicate that these tools do not seem so easy to maintain, which complicates adapting them to a specific context. We have presented some reasons that we believe may be behind this lack of maintainability. Furthermore, we have used the same measures to analyse our proposal and demonstrate that it is possible to develop another tool that seems to be easier to maintain and thus to customise for a specific context.

Part II

Background Information

Chapter 3

Enterprise Integration Patterns

To understand is to perceive patterns.
Sir Isaiah Berlin, British social & political theorist (1909–1997)

The enterprise application integration patterns documented by [Hohpe and Woolf](#) have been adopted as a cookbook for developing integration solutions. In this chapter, we first provide an overview of these author's work in Section §3.1. Section §3.2 presents the main categories of integration patterns they have documented. Section §3.3 illustrates how to combine the integration patterns. Finally, Section §3.4 summarises the chapter.

3.1 Introduction

An important contribution to the field of Enterprise Application Integration was done by Hohpe and Woolf [54] by means of their book on integration patterns. In this piece of work, the authors documented several patterns that software engineers can use to develop their integration solutions. These patterns revolve around the concept of message, which is an abstraction of an envelop that can be used to transfer data from an application into another, and even to invoke their functionality. Integration solutions that are based on messaging allows for asynchronous communication between applications, which makes them loosely coupled.

The integration patterns documented by [Hohpe and Woolf](#) can be considered as the first-step to establish a common vocabulary within the Enterprise Application Integration community, which is expected to result in domain-specific languages. Unfortunately, the patterns are described at a rather high conceptual level. Each one was given a name, a description of the context in which it can be used, and a description of how to solve a specific problem.

This catalogue of patterns has inspired important software tools that are currently available in the market of Enterprise Application Integration solutions. In the following sections we, introduce the main categories of integration patterns and give an example of how to use some patterns together.

3.2 Categories of patterns

In their book, [Hohpe and Woolf](#) documented sixty five integration patterns that were classified into six categories, namely: message construction, messaging channels, message routing, message transformation, messaging endpoint, and system management. Integration solutions developed using these patterns also follow the architectural pattern Pipes and Filters [44]. The pipes are supported by messaging channels and the filters by the remaining categories of integration patterns. Below, we describe each category.

Message Construction: Messages are containers of data that flow inside an integration solution. Roughly speaking a message consists of two parts, namely: a header and a body. The header holds meta-data about the data that

is carried in the body; it is the body that is expected to be modified, transformed, and routed through an integration solution.

The integration patterns in this category document the different types of messages that a software engineer may need to create, not only to transfer data amongst applications, but also to invoke functionalities, and send notifications. Furthermore, they document how to create messages to support request-reply communications and deal with situations in which a message must not be processed further since it can be considered stale.

Messaging Channels: Channels are part of the messaging infrastructure used to support the development of an integration solution, such as Java JMS [95] or Microsoft MSMQ [92]. They are used as resources to/from which messages can be written/read in total asynchrony. The writer and the reader can be either the applications being integrated or the integration solution. Simply put, a channel is a logical address that software engineers have to configure according to the adopted messaging infrastructure. A channel can be used by a single integration solution or can be shared by two or more solutions. Each messaging infrastructure may provide different types of channels and different configurations.

The integration patterns in this category document the use of channels for one-to-one and one-to-many communications, the setting up of a request-reply communication, how to restrict the type of messages a channels can receive, how to connect an application to the messaging system, how to deal with invalid messages or messages that have no readers, how to connect different channels in different messaging infrastructures, and so on.

Message Routing: Message routing comprises a set of integration patterns that allow to change the route of a message within an integration solution. The decision to which route a message has to go is usually made according to its contents. For this reason, the integration patterns have to inspect the body of a message; however, depending on the needs they can inspect the header as well. Some patterns can be configured with external contents, which are used to perform the routing of a message as well. An important characteristic of this kind of integration patterns is that they do not modify the contents of any messages.

The integration patterns in this category document how to route a message to a single or multiple destinations, how to define fixed or dynamic routing

policies, how to process individually each element from a list hold by a message, how to combine the results of individual processing of related messages so that they can be processed as a whole, how to remove unwanted messages from the workflow of an integration solution, and so on.

Message Transformation: When integrating applications, it is not usual that they use the same data model. Thus, the differences in data models usually require to transform the contents of messages from one format into another, so that they can be understood and processed by the applications that receive them. In addition to these application-specific data models, integration solutions may involve other applications that adopt standardised formats that are independent from an specific application, such as RosettaNet [96], HL7 [52], SWIFT [106], and HIPPA [51].

The integration patterns in this category document how applications that have different data models can be integrated, how data from one application can be sent to another application if the original message does not contain the required data, how to simplify the contents of a message, how to process messages that have equivalent contents but are represented in different formats, how to minimise dependencies when integrating applications that use different data models, how to create message formats that are independent from any specific application, and so on.

Messaging Endpoint: Since the applications in a software ecosystem are not usually designed with integration concerns in mind, it is not likely that they can send and/or receive messages. Therefore, software engineers have to develop messaging endpoints, which are pieces of code that interface an application and the integration solution, so that both can exchange messages. This piece of code has to be external to the application, since software engineers should preserve the applications unmodified.

The integration patterns in this category document several ways to interface an application and the integration solution to support communicating with one another. This may include interfaces that allow to compete for reading data from an application, to be selective when reading data, to provide an event-driven or a pooling communication type, to provide transactional support between an application and the integration solution, to map objects into messages, and so on.

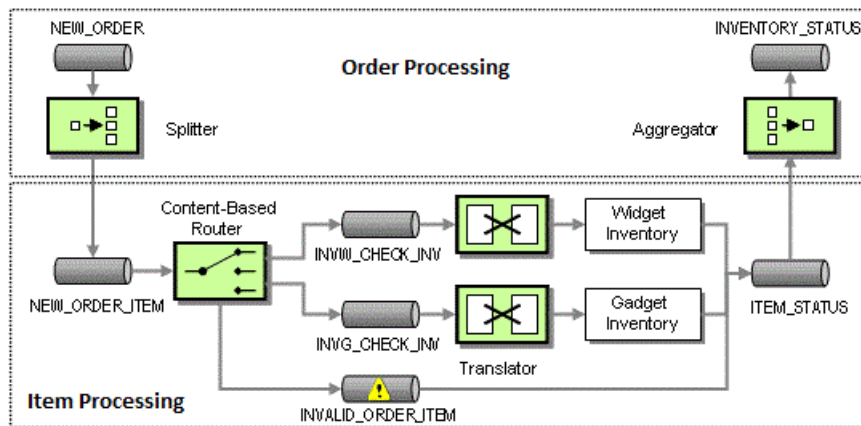


Figure 3.1: Processing order items individually (from Hohpe and Woolf [54]).

System Management: Operating an integration solution that is running in production is a challenging task. An integration solution may process thousands or even millions of messages exchanged amongst several applications that may have their state changed by every message. Furthermore, there can be performance bottlenecks not only in the integration solution, but also in the applications being integrated due to the communication with the integration solution. To make things even more challenging, the parts involved in an integration solution communicate asynchronously, may be distributed within the software ecosystem, and may fail.

The integration patterns in this category document different ways to manage an integration solution. They document how to detect if a building block is failing, how to debug them, how to inspect a message without affecting its regular processing, how to track messages, and so on.

3.3 An example

Processing orders is a very common task. In some cases, every item from an order has to be processed individually because, because they are served by different inventories. Figure §3.1 depicts the workflow and the integration patterns involved to check the availability of order items using Hohpe and Woolf's original notation.

The order processing starts by reading new orders from the `NEW_ORDER` channel and ends by writing messages with information regarding their availability to the `INVENTORY_STATUS` channel. Every message with a new order has to be split into individual messages, each of which must contain only one item; the resulting messages are written to the `NEW_ORDER_ITEM` channel. Software engineers have to route the messages to the `Widget Inventory` or the `Gadget Inventory` depending on their contents. The messages with items that do not belong to any of these inventories, are routed to the `INVALID_ORDER_ITEM` channel. Since the inventories have a different data model to represent an item, messages have to be translated into the appropriate data model using a message transformation pattern. The inventories write to the `ITEM_STATUS` channel the responses regarding the availability of items. The responses that correspond to items of the same order, including possible correlated invalid messages, are aggregated into a single message.

3.4 Summary

In this chapter, we have introduced the main categories of integration patterns that were documented by Hohpe and Woolf [54]. Each category deals with a group of conceptual patterns that software engineers can use to develop their integration solutions. We also reported on an example that provides an overall idea of some integration patterns and Hohpe and Woolf's original notation.

Chapter 4

Camel

*Think like a wise man,
but communicate in plain language.*

William B. Yeats, Irish dramatist & poet (1865-1939)

Camel is an open-source tool that is provided by Apache Software Foundation to support the design and implementation of Enterprise Application Integration solutions building on integration patterns. In this chapter, we first provide an overview of this tool in Section §4.1. Section §4.2 presents the building block used to wrap the messages that flow in an integration solution. Section §4.3 introduces the building blocks that are used to connect applications from the software ecosystem to the integration solution. Section §4.4 describes the building blocks that execute business integration logic in a workflow. Section §4.5 describes the building blocks that are necessary to design and implement the workflows. Section §4.6 introduces how errors are detected by Camel. Section §4.7 shows the design of the Café case study using the graphical notation of Camel. Finally, Section §4.8 summarises the chapter.

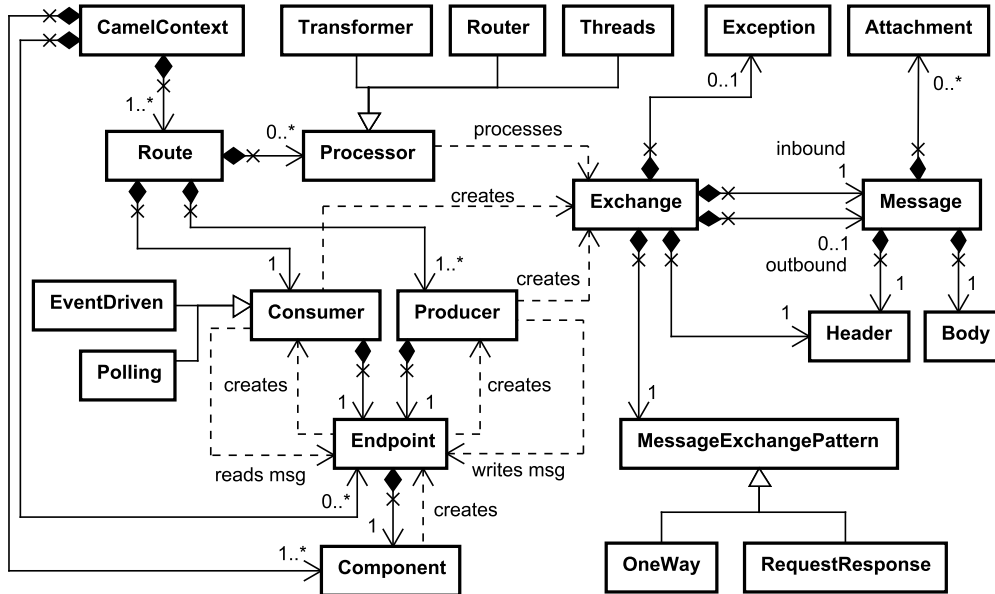


Figure 4.1: Conceptual model of Camel.

4.1 Introduction

Camel [58] is a Java-based software tool that aims to provide an integration framework with a fluent API [31] to support the design and implementation of Enterprise Application Integration solutions based on integration patterns. It was designed to be used by means of a Java- or a Scala-based domain-specific language, or by means of declarative XML Spring-based configuration files. The Java-based domain-specific language approach is the most popular in the Camel community. Camel is an open source tool that is hosted by the Apache Software Foundation. FuseSource is the company that provides products based on Camel, which includes a commercial version of Camel, a web-based graphical editor, and an Eclipse-based IDE with a graphical editor.

Central to the Camel architecture are the concepts of exchange, endpoint, processor, and route. The conceptual model in Figure §4.1 shows these concepts and their relationships. Exchanges are containers of messages. They flow inside an integration solution and carry messages from one processor to another. The messages contain data that endpoints read/write from/to

the applications available inside a software ecosystem, from one processor to another. Processors execute atomic integration tasks on messages and are chained in routes, which represent complex integration tasks.

4.2 Exchanges

Exchanges are building blocks that wrap inbound and outbound messages. Every exchange must be set to a message exchange pattern, which can be either one-way or request-response. The former indicates that the integration solution does not produce a response at the end of the workflow. On the contrary, the latter pattern indicates that the integration solution returns a response. Processors consider the inbound message in the exchange for their processing. The result of the processing can be stored back in the inbound message or as an outbound message. If a processor stores a message in the outbound message of an exchange, Camel transfers the outbound message to the inbound message before passing the exchange to the next processor in the workflow. At the end of the workflow, if an exchange holds an outbound message and it is set with a request-response pattern, then Camel uses this outbound message to produce a response; otherwise, if the pattern is set to one-way, the outbound message is thrown away.

Messages have a header, a body, and attachments. The header contains meta-data information that is associated with the message, which is an arbitrary piece of data that can be used during the processing of a message. Headers are implemented as a map that stores data in the form of name-value pairs, which are referred to as attributes. The body allows to store the main data contents of a message. Both, header and body can be read or modified at any time during the workflow. Attachments allow messages to carry additional data that goes through the solution without further processing.

Similarly to messages, exchanges also have a header. The difference with regard to a message header is that it aims to store global-level data. This information is available to all processors in the integration solution, independently from the inbound and outbound messages an exchange wraps. Camel uses this header to store information about the protocol being used to read the data in the corresponding message from an application, such as the encoding type, address, security permissions, and data that is related to service-level agreements. Note that when a processor creates a message, it does not contain the headers or the body of the message from which it originates unless the soft-

ware engineer copies them explicitly. We provide additional details about processors in Section §4.4. Any `Exception` that occurs during the processing of a message is captured by Camel and stored in the exchange, so that information is available to be used in error recovery.

4.3 Endpoints

Endpoints are building blocks used at the beginning and the endings of the integration solution workflow. They are used to connect applications from the software ecosystem to the integration solution. Endpoints are created from components, which are responsible for implementing the low-level transport protocol necessary to read/write data from/to a particular resource. Camel provides an extensive list with more than 80 different types of components, including components for files, databases, e-mail systems, queues, enterprise java beans, remote method invocations, Amazon's simple storage service, HL7, LDAP, RSS, HTTP, and SIP.

Every endpoint provides an interface that allows to create consumers and producers of messages from/to endpoints. They provide a high-level interface that software engineers can use to perform read/write operations; however the semantics of these operations depend on the type of component used to create the endpoint.

When a consumer uses an endpoint to read from an application, it creates an exchange to wrap a message that contains the input data, adds information about the resource to the header of the exchange, and feeds the exchange into the route. There are two types of consumers, namely: event-driven, which provide an interface on which clients can invoke methods, and polling consumers, which are consumers that have to poll an application periodically to gather data from it, e.g., a folder or a database. A producer, receives an exchange and writes the inbound message to the application.

4.4 Processors

Processors represent the processing units inside the workflow of an integration solution. They are building blocks that can transform and route exchanges in a workflow. Transformers are processors that change the payload

from one format to another. Routers are applied to change the trajectory of messages in the workflow based on a user-defined criterion.

Threads are a particular type of processor. They are used to define a pool of threads in a certain point in the workflow to enable concurrency from this point onwards. The processors after this point are executed using threads from this thread pool. Camel allows software engineers to use this strategy to speed up the performance of the integration solution in those parts of the workflow that consume more system processing. Unfortunately, using a threads processor breaks transaction boundaries, *i.e.*, if such a processor is used, then transactions are not preserved.

4.5 Routes

Routes represent workflows inside integration solutions. Roughly speaking, a route is composed of a consumer, zero or more chained processors, and one or more producers. Every exchange a consumer creates is processed by the chain of processors preceding the target producers. If a route does not include any processor, then the route implements a simple bridge pattern [54] that reads data from an application and writes it to other(s).

The Camel context has a global view of the types of components available, the endpoints and routes that are created, and it is responsible for managing the execution of routes, *i.e.*, it acts as the Runtime System of Camel. Every route involved in an integration solution has to be registered to the context, differently from the endpoints and components, which are automatically managed by the context. Although an integration solution can have several routes, every route is independent from each other, which means they can only exchange data by means of endpoints. In this case a producer from a route writes to an endpoint from which a consumer of another route reads.

Consumers are executed with their own pool of threads provided by Camel. By default, the same thread that is allocated to execute a consumer executes the whole into which it feeds messages. In this scenario, at the end of the route, if the exchange has a request-response message exchange pattern, a response is returned to the application that has activated the consumer. This response is given back using the same thread that has executed the consumer and consequently the whole route is executed synchronously. In scenarios in which a route includes a threads processor, when the execution of the work-

flow reaches it, the remaining execution can be as follows: a) if the current exchange has a one-way message exchange pattern, the current thread is released and the execution follows with a new thread; b) if the current exchange has a request-response message exchange pattern, the current thread remains blocked until the new thread finishes the processing of the remaining route. When this happens, the blocked thread is used to return a response to the particular application that has activated the consumer, unless the consumer's endpoint allows for asynchronous request-response.

4.6 Error detection

In Camel, when an exchange cannot be processed, an exception is raised. Camel provides software engineers with two mechanisms to detect errors. The first uses try-catch constructors, which have to surround the code that can potentially fail. The second is more sophisticated and allows to configure an error handler based on a redelivery strategy and a dead letter channel, to which exchanges that have failed and cannot be redelivered are moved. By default, a dead letter channel is just a logger of errors, but it can be configured as a queue that stores exchanges that have failed, so that they can later be read from this queue. An error handler can be configured on a global or per-route basis. At the global level, it gets the exceptions from every route, applies the same redelivery strategy, and uses the same dead letter channel independently from the route. Contrarily, if configured at the route level, the error handler allows for different redelivery strategies and dead letter channels.

4.7 The Café integration solution

Figure §4.2 shows the design of the Café case study using Camel's graphical notation. Cf. Section §12.2 for further information on this case study.

The integration solution was implemented using five routes that communicate by means of internal queues. The workflow starts at route (a). In this route, endpoint (1) is used to read orders from queue `direct:orders`. Every order is passed on to splitter (2), which breaks them up and generates new messages for every drink item in the order. The new messages that contain the drinks are written to the internal queue `direct:drinks` by means of endpoint (3). Route (b) was designed to read messages from `direct:drinks`,

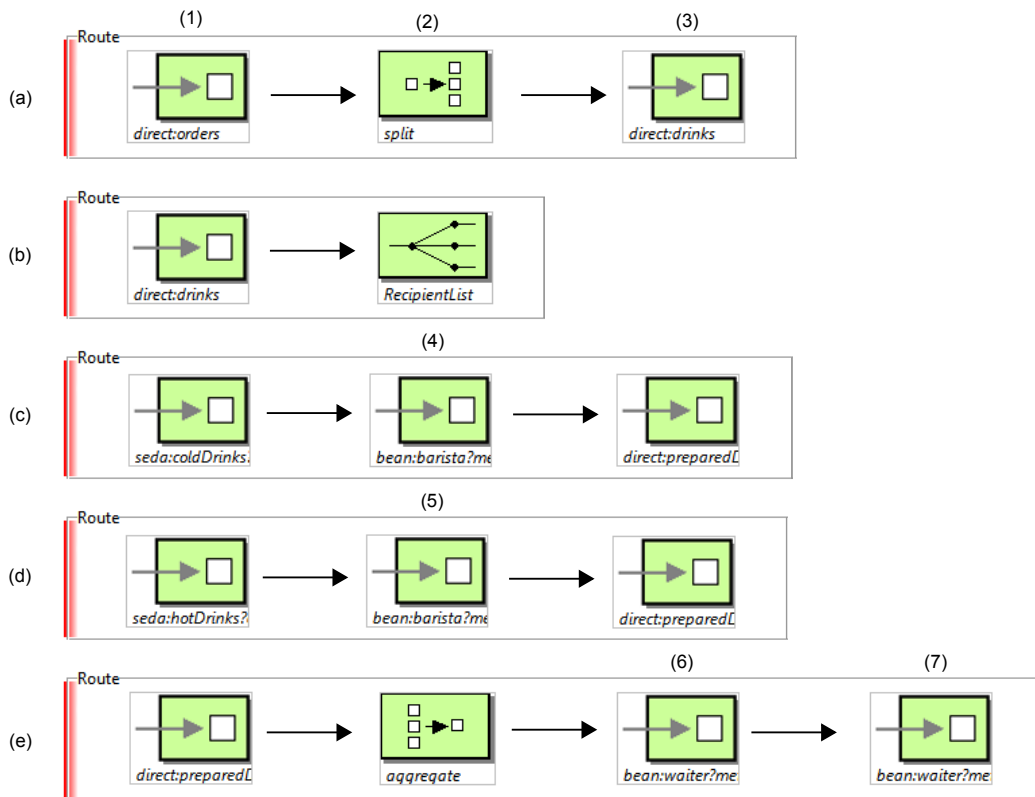


Figure 4.2: *Café integration solution designed with Camel.*

and, based on their contents, its recipient list routes messages whether to `seda:coldDrinks` or `seda:hotDrinks` internal queues. Recall that every route has always a producer endpoint, however in this route they are not shown in the graphical notation because the Java beans that implement the recipient list routing policy also act as producers by writing their response directly to the `seda:coldDrinks` and `seda:hotDrinks` queues. Routes (c) and (d) execute in parallel. They read messages from the `seda:coldDrinks` and `seda:hotDrinks` queues, respectively. The communication with the cold and hot baristas is done by means of endpoints (4) and (5), respectively. Their responses are sent to the `direct:preparedDrinks` internal queue, which is then the input for route (e). This route has an aggregator that aggregates all drink items from the same order into a single list, which is sent to endpoint (6). This endpoint builds a delivery message for the list of items, which is sent to endpoint (7), so that delivery messages are written to an application.

4.8 Summary

In this chapter, we have introduced Camel, which is an open-source Java-based software tool provided by the Apache Software Foundation to support the design and implementation of Enterprise Application Integration solutions. Furthermore, we have described the concepts of exchange, endpoint, processor, and route, and how these concepts are related one to each other. These are the most relevant concepts in the architecture of Camel. We also gave an introduction to how Camel deals with errors that can occur during the processing of an exchange. We have also reported on how to model the Café case study using Camel. The implementation is carried out by means of a fluent API, and can be assisted by an Eclipse-based IDE with a graphical editor.

Chapter 5

Spring Integration

*The limits of my language are the limits of my world.
Ludwig J.J. Wittgenstein, Austrian philosopher (1889-1951)*

Spring Integration is an open-source tool supported by VMware Inc., and extends the Spring Framework to support the design and implementation of Enterprise Application Integration solutions based on integration patterns. In this chapter, we first provide an overview of this tool in Section §5.1. Section §5.2 presents the building block used to wrap data that flows and is processed in an integration solution. Section §5.3 introduces the building blocks that execute business integration logic in a workflow, and allow to interact with the applications from the software ecosystem. Section §5.4 describes the building blocks used to transfer data inside an integration solution. Section §5.5 introduces how errors are detected by Spring Integration. Section §5.6 shows the design of the Café case study using the graphical notation of Spring Integration. Finally, Section §5.7 summarises the chapter.

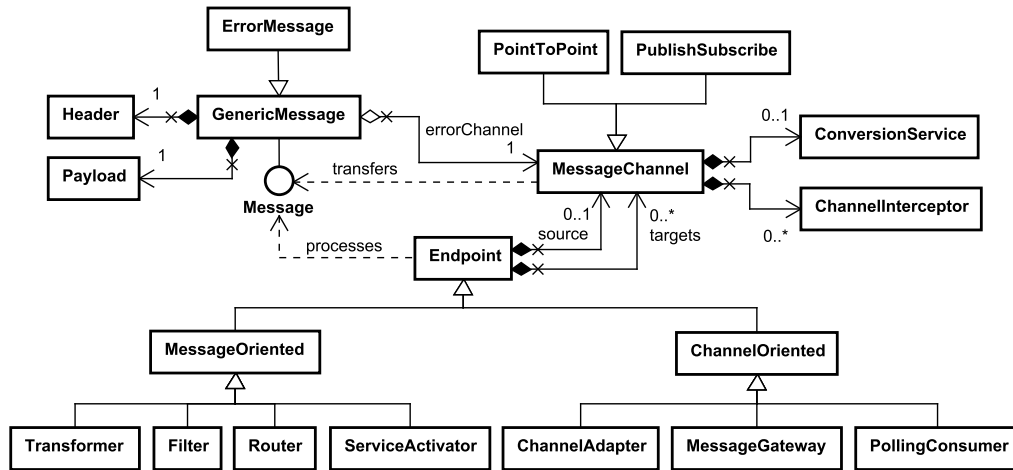


Figure 5.1: Conceptual model of Spring Integration.

5.1 Introduction

Spring Integration [30] is a Java-based software tool built on top of the Spring Framework container. It aims to extend this framework to support the design and implementation of Enterprise Application Integration solutions. Following the philosophy of Spring Framework, Spring Integration promotes the use of XML Spring-based files to configure integration solutions, although it is also possible to use Spring Integration as a command-query API [31]. Spring Integration is an open source tool that includes an Eclipse-based IDE with a graphical editor. The tool is led and supported by SpringSource, a division of company VMware Inc. VMware does not commercialise an enterprise version of Spring Integration, instead they use individual Spring Integration components in their commercial tools, such as vFabric RabbitMQ and vCenter Orchestrator.

The architecture of integration solutions implemented with Spring Integration have to follow the Pipes-and-Filters design pattern [44]. In this pattern, messages flow through several independent processing units (filters) that are communicated by means of channels (pipes). Messages are implemented with a building block with the same name, filters are implemented with endpoints, and pipes are implemented with message channels, cf. Figure §5.1.

5.2 Messages

Messages wrap data that flows and is processed in an integration solution. Spring Integration defines a general interface for messages that aims to provide access to the header and the payload of a message. The header allows software engineers to add/read meta-data information associated with the message, and is implemented as a map that stores data in the form of name-value pairs, which are referred to as attributes. There is not a limit for the number of attributes neither a limit for the size of the meta-data stored in an attribute; however, once a message has been created its header cannot be changed, since it is immutable. The payload allows to store the contents of a message, which can be read and modified within the workflow. Although the API of Spring Integration is based on the command-query style, it provides a fluent API [31] to create messages.

There are two implementations for the message interface, namely: generic message and an error message. The former represents regular messages that flow in an integration solution in normal conditions, whereas the latter represents messages that are created by Spring Integration when an error occurs during the processing of a regular message. To report eventual errors, generic messages are configured, by default, with a general error channel to which error messages are sent. This configuration can be changed by software engineers, so that error messages can be redirected to a different channel. We provide additional details about channel types in Section §5.4. Error messages are only created by Spring Integration and the difference between this type of message and a generic message is that the payload of the former must have an object of class `Throwable`, whereas the latter may have an arbitrary object of an arbitrary type.

5.3 Endpoints

Endpoints are building blocks that read, process, and write messages. They must always be connected to at least a source or a target channel. Roughly speaking endpoints can be grouped into message or channel-oriented endpoints. Message-oriented endpoints focus on performing a task on a message, possibly changing its contents. Endpoints in this group can be classified as transformers, filters, routers, or service activators.

Transformer endpoints aim to change an inbound message by transforming its payload from one format into another (e.g., from an XML document into a String), or by adding or removing contents to/from it. Filters apply a filtering policy, usually taking into account attributes in the header or the body, to evaluate whether a message can continue in the workflow of an integration solution or it has to be dropped. Routers are used to decide to what channel(s) an inbound message should be written, to aggregate or split messages. Service activators are a very generic type of endpoint. They aim to wrap an arbitrary object as a service, so that messages in the workflow can be arbitrarily processed. A source channel is used to send messages to the service, and if the service returns a value, this is done by means of a target channel.

Channel-oriented endpoints focus on providing support to bridge the communication between applications and integration solutions, or provide functionality to access the internal channels of an integration solution. Endpoints in this group can be classified as channel adapters, message gateways, or polling consumers. Channel adapters are responsible for reading/writing data from/to a particular type of resource. Their interface provides software engineers with a layer of abstraction on top of the low-level transport protocols necessary to read/write. Spring Integration provides several types of channel adapters, including files, databases, queues, web services, FTP servers, remote procedure calls, remote objects, HTTP servers, instant messaging systems, and social networks protocols. Message gateways aim to communicate with applications, however, they are used to provide a proxy that applications can use to push data to the integration solution. This endpoints enable clients to work with objects instead of messages, since they can push objects to the endpoint and the message gateway is responsible for wrapping them into messages and write the results to the appropriate channels, or vice-versa.

Endpoints can write messages to a target channel, independently from the type of channel and how messages are transferred by the channel. Contrarily, reading messages depends on how messages are transferred. Roughly speaking, they can be transferred synchronously or asynchronously. Channels transfer messages synchronously by default, which means that messages are read by and endpoint as they are written by the previous endpoint. Pooling consumers come into the picture when a channel that transfers messages asynchronously is used. In this case a polling consumer endpoint is necessary to check the channel for new messages. As long as endpoints are communicated by synchronous channels, they are executed in the same thread; contrarily, endpoints that communicate by means of asynchronous channels can execute on different threads. In the latter case, the thread associated with the endpoint

that writes the message to the asynchronous channel is released immediately after the endpoint completes the writing operation.

5.4 Message channels

Channels are responsible for transferring messages between endpoints. By default channels do not put any restriction on the messages they transfer, however they can be configured to accept messages with only certain type(s) of payload. If a message with another type is received, then Spring Integration attempts to convert the payload to an acceptable type using a conversion service, either built-in or user-defined. If no conversion service is configured or the conversion fails, an `Exception` is thrown. Every channel can also define zero or more interceptors. Channel interceptors allow to intercept messages that are read or written a channel without altering the workflow. This is an interesting approach for debugging and monitoring integration solutions.

Message channels can be classified along two axes: whether they deliver messages to a unique endpoint or not, and whether they are synchronous or not. Depending on the number of readers, message channels can be classified into point-to-point channels, in which there is a unique reader, and publish-subscribe channels, in which there can be an arbitrary number of readers. Synchronous channels require a writer and a reader to be ready simultaneously so that a message can be transferred through them; depending on whether the writer and the reader execute on the same thread or not they can be further classified into direct channels and rendezvous channels. Asynchronous channels, on the contrary, decouple the thread that writes a message to them from the thread that reads a message from them. Asynchronous channels may be unbounded or bounded, and they can optionally deliver their messages using a user-defined priority criterion.

5.5 Error detection

Endpoints can raise an exception during the processing of a message. Spring Integration allows software engineers to configure an `error-channel` attribute to an endpoint. Thus, when an exception is raised, the Spring Integration detects this exception, wraps it with an `ErrorMessage`, and sends the `ErrorMessage` to a channel configured to receive the errors. If there is not such

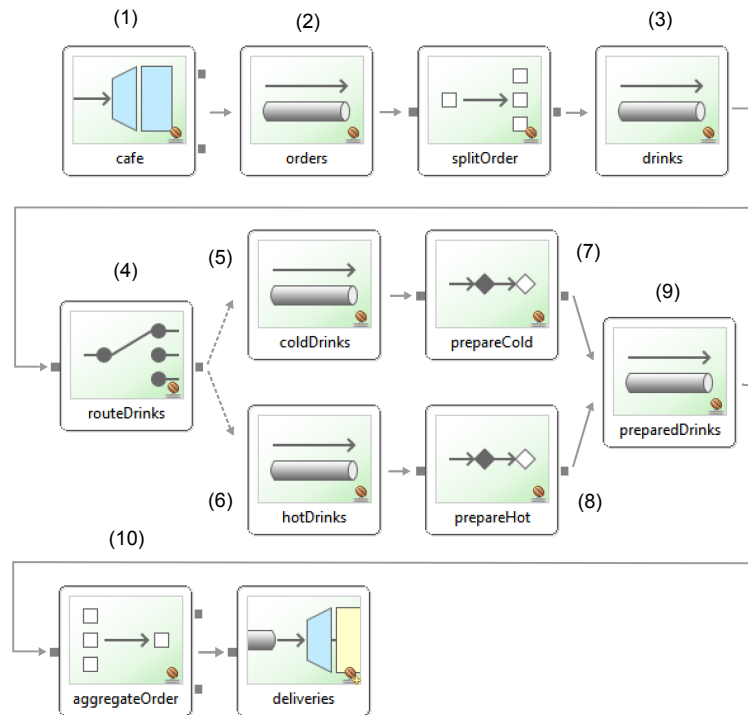


Figure 5.2: *The Café integration solution designed with Spring Integration.*

channel, then an exception is thrown in the code and software engineers have to capture them with a traditional Java try-catch block in the Java source code.

5.6 The Café integration solution

Figure §5.2 shows the design of the Café case study using Spring Integration’s graphical notation. Cf. Section §12.2 for further information on this case study.

The workflow of this integration solution starts at message gateway (1). This endpoint allows to receive orders from external resources and writes them to channel (2), which is used to transfer the orders to the next endpoint, a splitter. The splitter breaks them up and generates new messages for every drink item in the order. Channel drinks (3) is used to transfer these new

messages to router (4), which has to inspect every message in order to route them either to the cold drinks channel (5) or to the hot drinks channel (6). These channels are used to communicate with service activators (7) and (8), which then interact with external Java beans that implement the baristas that are responsible for preparing the cold and hot drinks. The responses from the baristas are sent to channel (9), which acts as a merger for the flow. Endpoint (10) is an aggregator that builds deliveries by aggregating all drink items from the same order into a new message. The last endpoint is a channel adapter used to write messages to an external resource.

5.7 Summary

In this chapter, we have introduced Spring Integration, which is an open-source Java-based software tool provided by VMware Inc. to support the design and implementation of Enterprise Application Integration solutions. Furthermore, we have described the concepts of message, endpoint, and message channel, and how these concepts are related to one another. These are the most relevant concepts in the architecture of Spring Integration. We also gave an introduction to how Spring Integration deals with errors that can occur during the processing of a message. We have also reported on how to model the Café case study using Spring Integration. The implementation is carried out by means of a command-query API, and can be assisted by an Eclipse-based IDE with a graphical editor.

Chapter 6

Mule

*Language shapes the way we think,
and determines what we can think about.*

Benjamin L. Whorf, American linguist (1897-1941)

Mule is an open-source tool provided by MuleSoft Inc. to support the design and implementation of Enterprise Application Integration solutions based on integration patterns; it builds on the concept of enterprise service bus. In this chapter, we first provide an overview of this tool in Section §6.1. Section §6.2 presents the building block used to wrap data that flows and is processed in an integration solution. Section §6.3 introduces the building blocks that are used to connect applications from the software ecosystem to the integration solution. Section §6.4 describes the building blocks that execute business integration logic in a workflow. Section §6.5 describes the building blocks necessary to design and implement the workflows. Section §5.5 introduces how errors are detected by Mule. Section §6.7 shows the design of the Café case study using the graphical notation of Mule. Finally, Section §6.8 summarises the chapter.

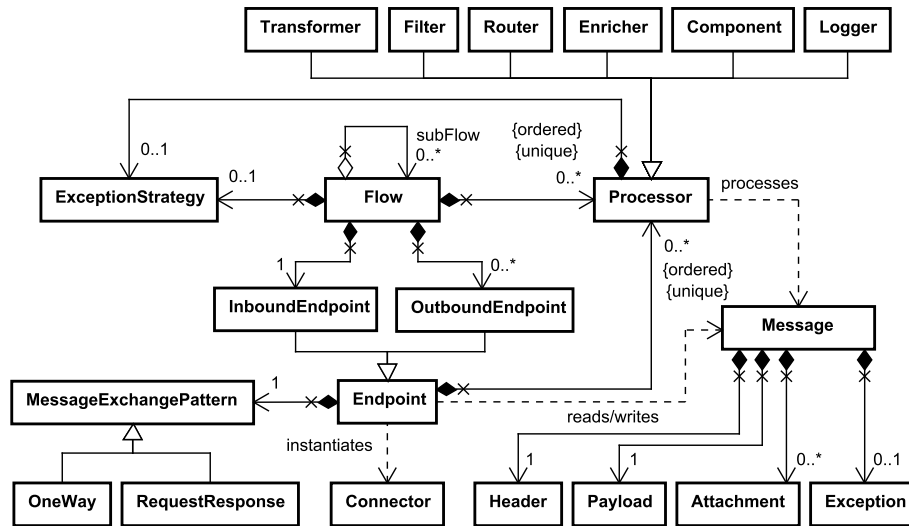


Figure 6.1: Conceptual model of Mule.

6.1 Introduction

Mule [27] is a Java-based software tool whose architecture is inspired by the concept of enterprise service bus. It aims to support the design and implementation of Enterprise Application Integration solutions based on integration patterns. It was designed to be used by means of a command-query API [31] or declarative XML Spring-based configuration files. The latter seems to be the most popular and recommended approach by the Mule community. Mule is open source and provides a community version that includes an Eclipse-based IDE with a graphical editor. A commercial enterprise version is also available and maintained by MuleSoft Inc., which supports the Mule project.

Central to the Mule architecture are the concepts of message, endpoint, processor, and flow. Figure §6.1 shows a conceptual model that describes the relationships amongst these concepts and other elements around them. Messages encapsulate the data that endpoints read/write from/to the applications available within a software ecosystem. On reading data, the corresponding message can be processed by a series of processors that, in the end, write them to one or more applications.

6.2 Messages

Data that flows in a Mule integration solution are wrapped into messages. Every message has a header that allows software engineers to add/read meta-data information associated with the message. The header is implemented as a map that stores data in the form of name-value pairs that are referred to as attributes. There is not a limit to the number of attributes neither a limit to the size of the meta-data stored in an attribute. The main data contents of a message is stored in its payload, which holds an `Object`. Different from the header, data in the payload can be read and modified during the processing of a message in a workflow. Additional data that is not usually intended to be processed, but needs to be kept in order to produce an output message, should be carried as attachments. Messages also define an exception element to hold an exception that occurred during the processing of the message, so that this information is available to be used in error recovery.

6.3 Endpoints

Endpoints represent inbound and outbound points in an integration process. They correspond to a specific instance of a connector. Connectors abstract away from the technical details to deal with low-level transport protocols, which carry out the interactions with a particular type of resource. The most common types of transport protocols are supported by Mule, which provides a list of connectors that range from connectors to files, databases, queues, web services, to connectors for social networks, cloud infrastructures, and business process management systems. Endpoints provide software engineers with a unique interface to read/write messages from/to a variety of applications. By default, Mule creates a pool of threads for every endpoint, so that an endpoint can handle several read/write operations at the same time.

Both endpoints and connectors have properties that software engineers can use to configure them. Generally, this configuration is a tradeoff. A connector can be configured mixing properties regarding reading or writing operations, so that it is not necessary to have different connectors for each operation. Thus the kind of use (read/write) depends on the endpoint that uses it. On the contrary, it is necessary to configure an independent endpoint for each operation, although they can share the same connector. Although this way of configuration seems intuitive, it constraints the reuse of a connector. The reason is

that if the connector is configured for both operations, it must have information about the applications from which it has to read or write and is tightly coupled with the applications. Thus, to make connectors more reusable, it is necessary to configure independent connectors for each operation and take the information about the applications to the corresponding endpoints.

Software engineers can also configure processors to execute specific tasks inside endpoints. We provide additional details in Section §6.4. These tasks are intended to transform a message from a resource-specific format to a canonical format that is specific to an integration process, or to filter unwanted messages. In situations like these, the use of such processors aims to separate the wrapping logic for a particular resource from the integration process logic.

Endpoints support two kinds of message exchange patterns, namely one-way and request-response. The former indicates that the endpoint does not return a response when the message that has been read or written has been completely processed by the integration process. On the contrary, the latter pattern indicates that the endpoint returns a copy of the current outbound message. Not every endpoint supports both kinds of patterns, instead the support is constrained by the type of the connector that is used by the endpoint. For example, file connectors only support the one-way message exchange pattern, however HTTP connectors support both message exchange patterns (request-response by default).

6.4 Processors

Processors are building blocks that receive messages and do some processing with them. Every processor implements a small, atomic integration task taking into account the header, the payload, and/or the attachments of a message. Processors can be chained together to implement complex tasks that require several different types of processing on a message.

The processors supported by Mule can be organised into: transformers, filters, routers, enrichers, components, and loggers. Transformers are processors that change the payload from one format to another; filters can selectively filter some messages out of a workflow; routers are applied to change the trajectory of messages in a workflow based on a user-defined criterion; enrichers are used to add contents from external sources to a message; components allow to wrap objects to re-use functionality; finally, loggers write messages to

a log system. Every category also provides a general implementation that can be extended and customised by software engineers according to their needs.

Transformers and filters are quite common in integration solutions, which is the reason why Mule allows to configure global transformers and filters. This is interesting in situations in which the same kind of transformation or filter can occur several times in the same or even in different workflows.

There are many processor types that are fully-configured by default. This is common for very simple tasks chiefly in the transformers category, such as the transformation from an `Object` into its XML representation, from a byte array to an `Object`, or from `String` to `Object`. However, other processors in these categories as well as in the filters, routers, and enrichers categories, can have their integration logic modified by means of scripting languages. The language used depends on the type of message: one can use XPath for XML messages, or OGNL, JXPath or Groovy for Java objects.

6.5 Flows

Flows in Mule are used to implement integration processes. They chain together endpoints, processors, and other sub-flows. A flow includes one inbound endpoint, zero or more processors, zero or more outbound endpoints, and an optional exception strategy. Flows that do not include a processor, implement a pass-through integration process that simply moves data from a source to a target (in this case, we assume that the flow includes an outbound endpoint). If a flow does not include an outbound endpoint, then the inbound endpoint must be configured with a request-response message exchange pattern or use a component processor that interacts with an external resource to write messages out of the flow. The exception strategy receives messages whose processing has failed.

As soon as a read operation terminates in an endpoint, the thread on which it runs is released. Then, the inbound message is made available to the first processor in the flow by means of an internal queue. By default, messages are processed synchronously in the chain of processors that compose a flow; therefore Mule defines a pool of threads to be used by these processors, as well. Software engineers can change this default processing model used in flows by defining asynchronous scopes that embrace all the flow or only part of it. Asynchronous scopes are sub-chains in which every processor in the

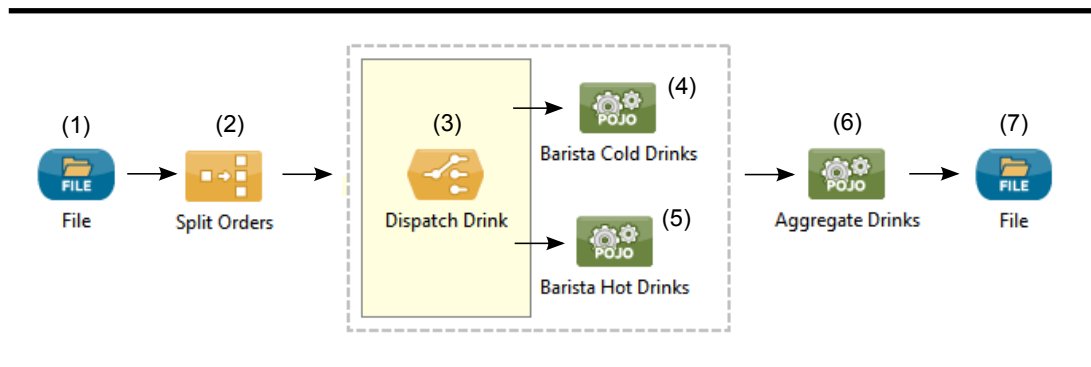


Figure 6.2: *The Café integration solution designed with Mule.*

chain runs in a different thread. Similarly to inbound endpoints, messages are made available to outbound endpoints by means of an internal queue.

Sub-flows are re-usable flows; the key is that they do not include endpoints; when they are invoked, the calling flow passes the current message to the sub-flow, waits for the response from it and then resumes processing. If the calling flow is executed synchronously, then the same thread running the calling flow runs the processors of the sub-flow for that message; otherwise the calling flow thread pool is shared with the sub-flow.

6.6 Error detection

In Mule, if a message cannot be processed, an exception is raised. Mule allows to configure an exception strategy object at the processor and/or flow levels. Thus, when Mule detects an exception, it logs it, adds it to the message that has failed, and forwards the message to the exception strategy object. An exception strategy is configured to use an `OutboundEndpoint`, so that the message can be stored into a resource, such as a queue or database.

6.7 The Café integration solution

Figure §6.2 shows the design of the Café case study using Mule’s graphical notation. Cf. Section §12.2 for further information on this case study. As of

the time of writing this dissertation, the graphical editor is in beta version and does not provide support for some integration patterns.

The workflow of this integration solution starts at file endpoint (1), which reads orders. Orders taken by processor (2) are split and generate new messages for every drink item. Then, the dispatcher processor (3) inspects every message in order to route them either to the Barista Cold Drinks (4) or to the Barista Hot Drinks (5). In this integration solution, processors (4) and (5) are interfaces that allow to invoke the business logic that implement the baristas. The outbound messages from these processors represent drinks that are prepared and then can be aggregated back into an order to which they correspond. Since the current version of the graphical editor does not support aggregators, we provide a ready-to-use aggregator, we have implemented the aggregation business logic in a separate Java class that is interfaced using processor (6). Finally, file endpoint (7) is used to deliver messages.

6.8 Summary

In this chapter, we have introduced Mule, which is an open-source Java-based software tool provided by MuleSoft Inc. to support the design and implementation of Enterprise Application Integration solutions. Furthermore, we have described the concepts of message, endpoint, processor, and flow, and how these concepts are related one to each other. These are the most relevant concepts in the architecture of Mule, which is inspired by the concept of enterprise service bus. We also gave an introduction to how Mule deals with errors that can occur during the processing of a message. We have also reported on how to model the Café case study using Mule. The implementation is carried out by means of a command-query API, and can be assisted by an Eclipse-based IDE with a graphical editor.

Chapter 7

Model-Driven Engineering

The purpose of science is not to analyse or describe but to make useful models of the world.
Edward de Bono, Maltese psychologist (1933-)

The Model-Driven Engineering discipline has been changing the way how software systems are built. It promotes models as first-class citizens in all phases of the software development process. In this chapter, we first provide an overview of this discipline and two possible approaches to carry it out in Section [§7.1](#). Section [§7.2](#) introduces the Model-Driven Architecture approach proposed by the Object Management Group. Section [§7.3](#) presents Software Factories, which is the approach proposed by Microsoft Corporation. Finally, Section [§7.4](#) summarises the chapter.

7.1 Introduction

The field of Software Engineering is involved in a paradigm shift that has important consequences on how software engineers construct and evolve systems. Central to this change is the Model-Driven Engineering discipline [100], which promotes models as first-class citizens in every phase of the software development process. Models are abstractions that allow software engineers to focus on the relevant aspects of a software system while ignoring details that are irrelevant. Behind this discipline is the idea to raise the level of abstraction of the overall development process, to capture systems as a collection of reusable models, to separate business logic descriptions from a particular platform implementation, and to automate the implementation phase [9, 49, 63, 69, 100]. There are several approaches to Model-Driven Engineering, including the Model-Driven Architecture [79] and Software Factories [47].

The Model-Driven Architecture was proposed by the Object Management Group and relies on their standard modelling languages to represent software systems using models at different levels of abstraction, and standard transformation languages to generate executable systems from these models. In this approach, UML [86] is the recommended language to represent the models, QVT [85] is the recommended language to write transformations between models at the same or different level of abstraction, MOF Model to Text [81] is the language used to generate executable code from a model, and, XMI is the recommended language to serialise models.

Software Factories was proposed by Microsoft Corporation and aims to integrate well-established areas of Software Engineering, such as product line families, domain-specific languages, frameworks, and patterns, into a new methodology and a set of tools that software engineers can use to develop software [47]. Software Factories does not target an architecture based on levels of abstraction or the use of standard languages. In fact, their adoption is up to the software engineers. To realise Software Factories, Microsoft provides a set of tools that build on .NET and Visual Studio [91]. IBM is another company that bets on Software Factories, but they target the Java platform and WebSphere [94].

In recent years, the use of domain-specific languages in the software industry is increasing very fast. Many authors argue that domain-specific languages bring important advantages over general-purpose languages, *e.g.*, they help raise the level of abstraction by providing language constructs that are very

close to the problem domain, they are smaller, easier for software engineers to learn and use, they are more expressive, they increase productivity, quality, and maintainability [4, 29, 31, 45, 107, 109]. The Software Factories approach is guided by the development and use of domain-specific languages. In the Model-Driven Architecture approach, although it focuses on the use of UML, which is a general-purpose language, it also gives the possibility to work with domain-specific languages. In such a case, the domain-specific languages are defined either as extension to UML, which are referred to as profiles [46], or as extensions to the MOF language [80]. As a result, the new language conforms to either the UML or the MOF syntax.

The Model-Driven Architecture focuses on applying Model-Driven Engineering to the design and implementation phases of the software development process. It provides a conceptual framework that software engineers can follow to generate software from models. Software Factories aims to support every phase of the software development process, by providing a methodology that integrates well-known areas in Software Engineering. Although they are different approaches, they can be complementary [19, 82]. In the following sections we provide additional details on each approach.

7.2 Model-Driven Architecture

The Model-Driven Architecture approach promotes the construction of software systems building on a set of inter-related views, at the same or different levels of abstraction. These views allow software engineers to separate business-oriented decisions from software design and implementation decisions. Each view corresponds to a model that describes the whole or a part of a system, with a focus on relevant details for that view. A model can be used as a source to produce other model(s) at the same or different level(s) of abstraction [77]. A typical situation within the same level of abstraction is to perform model refactoring, which aims to produce a new model with a better quality without changing the observable behaviour in the source model. When a model is used to produce other model(s) at a different level of abstraction, the target model includes more or less details regarding how it is implemented using a specific technology.

Transformation rules are written using transformation languages. Although QVT and MOF Model to Text are recommended by the Object Management Group, several other languages have gained importance in this field,

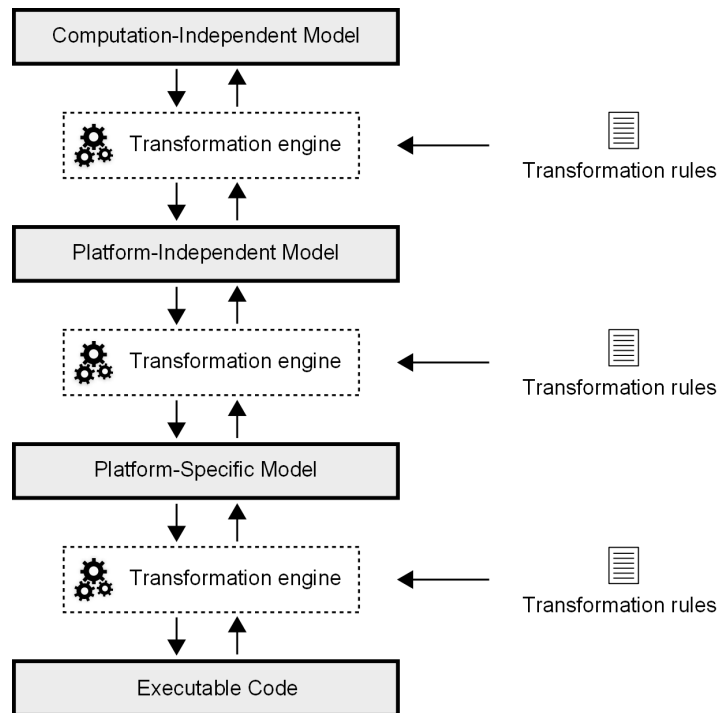


Figure 7.1: Abstraction levels in the Model-Driven Architecture.

such as ATL [62], MOFScript [84], and XSLT [65]. Rules and models are taken as input by transformation engines, which are responsible for producing other model(s) or the corresponding executable code. A transformation rule consists of a description of how one or more source elements can be transformed into one or more target elements. France and Bieman [32] classified transformations into two types: horizontal and vertical. The former corresponds to a transformation in which the source and target models are within the same level of abstraction; in the latter, the source and target models are at different levels of abstraction. Figure §7.1 shows the abstraction levels and their relationship.

The Model-Driven Architecture defines three levels of abstraction on top of the executable code of a software system, namely: computation-independent model, platform-independent model, and platform-specific model.

The computation-independent model is the highest level of abstraction. Models at this level aim to describe the main abstractions and functionalities

involved in a particular business domain. Business decisions shall not be influenced by an specific information technology, and shall use languages that provide constructs that are very close to the abstractions of the business domain. The resulting models are also referred to as domain models. An example of domain model can be the specification of which applications shall be integrated and which information they shall exchange to support a new business process.

A platform-independent model shall be obtained from the refinement of a computation-independent model. At this level of abstraction, models are refined to describe the operations, structure, and behaviour of a software system [10]. Information technology abstractions such as databases, communication channels, software patterns, component interfaces, and data structure, are introduced. However, there is not a commitment to a particular implementation technology. Models shall remain neutral, so that they keep software systems preserved from changes to the underlining technology, and can be reused several times to generate different platform-specific models [66, 79].

Platform-specific models are the lowest level of abstraction on top of the executable code. Models are obtained from the refinement of platform-independent models. Models at this level are bound with specific implementation technologies, such as a vendor specific database (e.g., Oracle, SQL Server, PostgreSQL, and so on), a communication protocol (e.g., HTTP, IIOP, RMI, and so on), or an application framework (e.g., Java EE, .NET, and so on). Consequently, the operations, structure, and behaviour in the models must include details on how they can be implemented in the chosen technologies. Transformations model-to-text are used to obtain an skeleton or a complete executable system from these models in a target general-purpose programming language.

7.3 Software Factories

Software Factories has been promoted as an approach towards to mass customisation of software [3, 47]. As a result, Software Factories targets a specific business domain in which a product line can be abstracted. A product line consists of a family of software artefacts that share core assets, including an architectural framework [18]. These assets can be systematically reused to reduce software development costs and shorten the time to market [67]. In Software Factories, the concept of asset includes configuration files, source code files, localisation files, build scripts, deployment manifests, test case def-

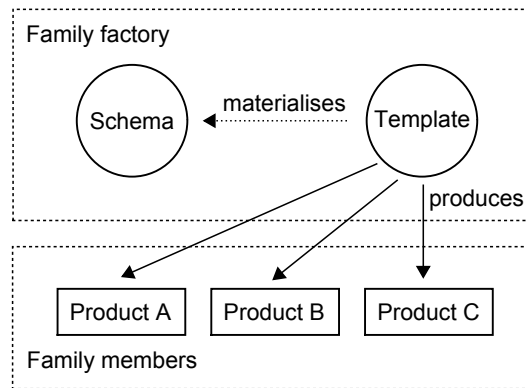


Figure 7.2: *Abstraction of a Software Factory.*

initions, models, transformations for automatic artefact generation, domain-specific languages, libraries, frameworks, patterns, how-to help pages, training material, and sample codes. During the life-cycle of a software factory, additional common assets can be identified across multiple products, and added to the factory.

Roughly speaking, Software Factories are organised into factory schema and factory template, *cf.* Figure §7.2.

A software factory schema is the meta-data of the factory, *i.e.*, it provides only a description of the factory and its contents. These descriptions also document the relationships amongst assets and includes guidelines regarding when and how they have to use these assets. The documentation of a relationship between two assets can include, for example, information on how to map an asset onto another. Furthermore, these descriptions can document which other assets shall be affected when changing a particular asset. Guidelines are valuable assets since they agglutinate much knowledge on how to do things in the best way. Thus, they are very important, chiefly for junior engineers, who can follow them, and senior engineers, who can refine the guidelines with their experience. Every asset has its roots on the business requirements of the product line family, which is also part of the schema. Descriptions in the schema can be divided into fixed and variable. The former represents assets that are reused exactly as they are in every product member of the family; the latter, allows for customisation to accommodate the particularities of each product.

Software factory templates are the materialisation of the schema into a set of actual artefacts and tools that are packaged and delivered to software engineers. These artefacts and tools have to be installed and integrated into the software development environment of the development team, so that actual software can be produced and delivered to clients. Thus, templates can be seen as a way to configure a domain-specific development environment to produce software with a uniform architecture, promote the reuse of domain assets, promote the automation of error-prone and recurrent tasks, to increase software quality, and reduce the costs involved in software production [47, 67].

7.4 Summary

In this chapter, we have introduced the Model-Driven Engineering discipline, which is changing the way how software engineers construct and evolve systems. Furthermore, we have described two approaches to carry out this discipline, namely: Model-Driven Architecture and Software Factories. The former was proposed by the Object Management Group and relies on their standard modelling languages to represent software systems using models at different levels of abstraction. The latter, was proposed by Microsoft Corporation and aims to integrate well-established areas of Software Engineering into a new methodology and a set of tools that software engineers can use to develop software.

Part III

Our Approach

Chapter 8

Domain-Specific Language

The function of modelling is to arrive at descriptions that are useful.
Richard W. Bandler & John Grinder, American authors (1950- & 1940-)

We describe our Domain-Specific Language to design integration solutions in this chapter. Section §8.1 introduces the concepts with which we deal to model integration solutions. Section §8.2 describes the abstract syntax of our language. Section §8.3 presents the graphical concrete syntax to represent the building blocks of our language. Section §8.4 introduces a general-purpose toolkit to support the design of integration solutions. Finally, Section §8.5 summarises the chapter.

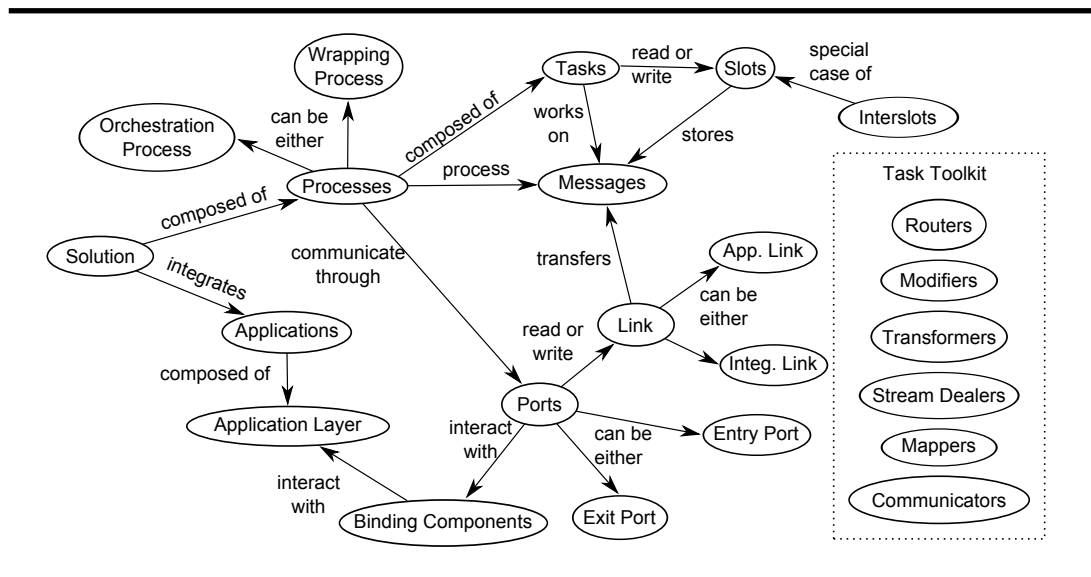


Figure 8.1: Conceptual map of our Domain-Specific Language.

8.1 Introduction

In this chapter, we describe our Domain-Specific Language, to which we refer to as Guaraná DSL. This language allows software engineers to design integration solutions using a graphical and very intuitive concrete syntax. Furthermore, Guaraná DSL can be used as a common and yet simple vocabulary for communicating in this field.

Figure 8.1 presents a conceptual map in which we introduce the concepts with which we deal to model integration solutions. The root concept is solution, which represents a collection of processes that co-operate to integrate a number of applications.

Processes serve two purposes, namely: there are processes that allow to wrap applications and processes that allow to integrate them. The former are reusable processes that endow an application with a message-oriented application programming interface that simplifies interacting with it. Implementing such a wrapping process may range from using a JDBC driver to interact with a database to implementing a scrapper that emulates the behaviour of a person who interacts with a user interface [22]. Orchestration processes, on the contrary, are intended to orchestrate the interactions with a number of wrapping processes and other orchestration processes.

Processes rely on tasks to perform their wrapping or their orchestration activities. Simply put, a process can be viewed as a message processor. A message is an abstraction of a piece of information that is exchanged and transformed across an integration solution. The structure of messages depends completely on the integration solutions in which they are involved.

Guaraná DSL provides a general-purpose task toolkit, which contains a collection of tasks that provide the foundations for many other special-purpose task toolkits for different integration contexts, such as RosettaNet-oriented tasks in business-to-business contexts [96], HL7-oriented tasks in health contexts [52], SWIFT-oriented tasks in financial contexts [106], or HIPAA-oriented tasks in insurance contexts [51], to mention a few. In Figure §8.1, we illustrate the main categories of tasks in the general-purpose task toolkit; *cf.* Section §8.4 for a comprehensive description.

Note that our proposal does not preclude several tasks (including several instances of the same task) from executing in parallel. This makes it impossible for tasks to communicate directly to each other. Instead, they communicate indirectly by means of slots. A slot acts as a buffer in-between tasks, *i.e.*, they allow a task to output messages that shall be processed asynchronously by another task.

Java Business Integration is an specification of a pluggable architecture of services [17] to which several Enterprise Service Buses have adhered, such as the Open ESB [90]. Typical Enterprise Service Buses provide so-called adapters, which are used to interact with the applications being integrated. In Java Business Integration, adapters are referred to as binding components. The binding components implement the low-level transport protocol necessary to carry out this interaction. There are many types of binding components available nowadays, which allow the integration solutions to connect to almost any existing application in a software ecosystem. The catalogue includes binding components for databases, local files, FTP, SOAP/HTTP, RSS, SMTP, RMI/IIOP, JMS, HL7, LDAP, DCOM, XMPP, SMPP, SNPP, S3, and so on. Our Domain-Specific Language allows to reuse the large catalogue of binding components provided by Open ESB.

Processes use ports to communicate with each other or with the applications involved in an integration solution. Simply put, the purpose of a port is to abstract away from the details required to interact with a binding component, which, in turn, abstracts away from the details required to interact with an application within the software ecosystem or another process. Binding components are used by means of a special kind of task, referred to as

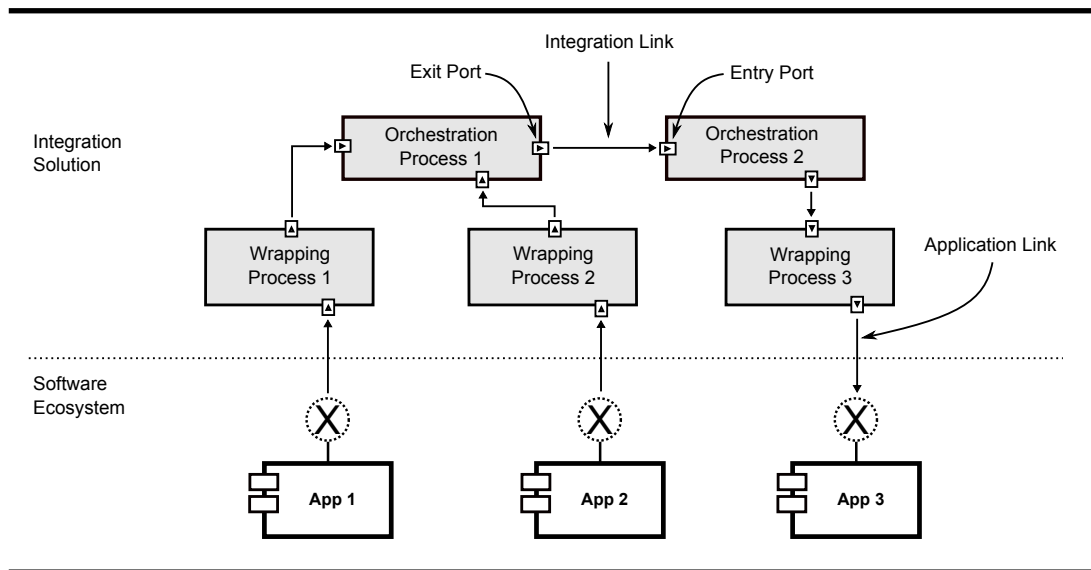


Figure 8.2: Typical integration solution designed with Guaraná DSL.

communicator. The interaction with an application occurs at one or more layers, *i.e.*, data layer, data access layer, business logic layer, and user interface layer. Ports can be either entry or exit ports, depending on whether they were designed to read messages from a process or an application, or to write messages to them.

Note that ports usually need to transform the messages they transfer, which implies that they are composed of tasks, as well. This means that they also need slots to help their tasks work as much asynchronously as possible. Another subtle implication is that there must be a slot to communicate a task in a port to a task in the process to which the port belongs. We refer to such slots as interslots.

Figure §8.2 shows, from an abstract point of view, a typical integration solution that was designed using the Guaraná DSL. The integration solution is composed of three wrapping processes that interact with the applications in the software ecosystem, and two orchestration processes that implement the integration business logic to be executed. In this integration solution, the workflow processes messages that are read from App1 and App2 and writes the results to App3.

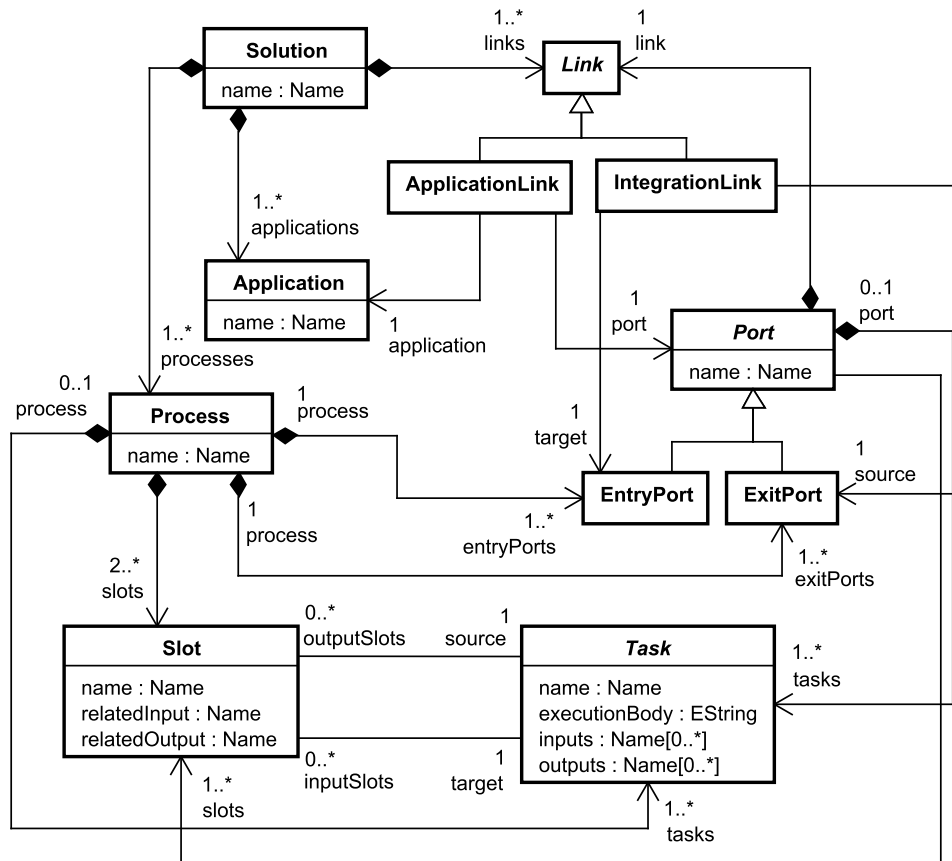


Figure 8.3: Main constructors of Guaraná DSL.

8.2 Abstract syntax

This section describes the part of our metamodel, aka abstract syntax, that is related to the core domain-specific language. This metamodel conforms to the Eclipse Modelling Framework [105], so that it can be used exactly as it is to develop domain-specific workbenches using Eclipse-based technologies. The constraints in the metamodel were validated using the Dresden OCL Toolkit [25] to ensure they conform to the Object Constraint Language specification. Figure 8.3 provides an overall picture to guide the reader through the following subsections.

8.2.1 Integration solutions

Solution is the root class of our metamodel, and it represents an integration solution. A Solution has a name property, which is used for documentation purposes only, and consists of one or more Processes, one or more Applications, and one or more Links. A Solution must fulfill the following invariants:

```
context Solution
  inv: applications->isUnique(name)
  inv: processes->isUnique(name)
  inv: links->isUnique(name)
```

They state that the names of the applications, processes and links must be unique. Note, however, that an application and a process may have the same name, since there are no chances to mistake them.

8.2.2 Processes

Class Process represents either a wrapping or an orchestration process. A Process is composed of at least one EntryPort, at least one ExitPort, at least one Task, and at least two Slots. A Process has to fulfill the following invariants:

```
context Process
  inv: tasks->union(entryPorts.tasks->union(exitPorts.tasks))->
      isUnique(name)
  inv: slots->isUnique(name)
  inv: entryPorts->union(exitPorts)->isUnique(name)
  inv: tasks->select(oclIsKindOf(Communicator))->size() = 0
  inv: let interslots: Set(Slot) = slots->select(s: Slot |
      not self.tasks->includes(s.source) and
      self.tasks->includes(s.target) or
      self.tasks->includes(s.source) and
      not self.tasks->includes(s.target)) in
      interslots->size() = self.entryPorts->size() +
                          self.exitPorts->size()
```

These invariants state that tasks, slots and ports must have unique names, that a process cannot contain any tasks of kind Communicator because these

tasks are specific to ports, and that there can be only one interslot per port.

To understand the first invariant, recall that both processes and ports can contain tasks, and they all must have different names. Thus, for each process, we need to calculate the set of tasks of which it is directly composed, union the set of tasks in its entry and exit ports.

Note, however, that the invariant regarding slots is slightly different, since we do not need to calculate the slots of a process, union the slots of its entry and exit ports; instead, we can simply write `slots->isUnique(name)`. The reason is that, at least in theory, interslots belong to both a process and a port; unfortunately, EMF does not allow to model such situations. The only solution is that property `slots` holds all of the slots involved in a process, including the slots in its entry and exit ports.

The last invariant also deserves an explanation. It states that there can only be one slot connecting the tasks that are contained in a port to the tasks that are contained in the corresponding process, *i.e.*, there can be at the most one interslot per port. The most difficult part of the invariant is the identification of interslots: they are calculated as the set of slots whose source task is not included in the set of tasks of which a process is directly composed, but the target is, or vice-versa. Note that if a source or a target task in a slot does not belong to a process itself, it must belong to one of its ports, which implies that the original slot is actually an interslot.

8.2.3 Ports and links

Ports are composed of tasks and slots, that get connected by Links; these, in turn, can be either `ApplicationLinks`, which connect `Applications` to `Ports`, or `IntegrationLinks`, which connect `EntryPorts` to `ExitPorts`. Recall, however, that the inability to represent interslots as shared objects prevented us from modelling the slots of which a port is composed as a proper containment property. Instead, we need to calculate the slots of which a port is composed by means of a derivation, namely:

```
context Port::slots: Set(Slot)
  derive:
    tasks->collect(outputSlots)->
      union(tasks->collect(inputSlots))
```

Another derivation is property link; recall that every port must be connected to a link so that messages can be transferred. Links can be seen as an abstract communication mechanism. The problem is that links should belong to both an integration solution and some of its ports, which is not possible. This is the reason why property link is also derived, namely:

```
context Port::link: Link
  derive:
    let appLink: Link = ApplicationLink.allInstances()->
      any(port = self) in
    let intLink: Link = IntegrationLink.allInstances()->
      any(source = self or target = self) in
    if not appLink.ocllsUndefined() then
      appLink else intLink
    endif
```

Note that the derivation is defined in class Port. This is the reason why the formula tries to find both an application and an integration link whose port is the current context; depending on whether the port is actually connected to an application or to a process, either appLink or intLink shall not be undefined.

Furthermore, ports must fulfill the following invariants:

```
context Port
  inv: tasks->isUnique(name)

context EntryPort
  inv: tasks->one(ocllsKindOf(Communicator))
  inv: tasks->one(ocllsKindOf(InCommunicator))

context ExitPort
  inv: tasks->one(ocllsKindOf(Communicator))
  inv: tasks->one(ocllsKindOf(OutCommunicator))
```

The previous invariants state that the tasks in a port must have unique names, that an EntryPort must have one Communicator of kind InCommunicator, and that an ExitPort must also have one Communicator of kind OutCommunicator. Whilst the former kind of communicator is used to read messages from a binding component, the latter is used to write messages to the binding component.

There is a final invariant: we do not allow for 'looping' processes, *i.e.*,

processes in which a port is connected to another port in the same process. To avoid this kind of anomaly, we introduced the following invariant in our metamodel:

```
context IntegrationLink:
  inv: not (source.process = target.process)
```

8.2.4 Tasks and slots

Every task has a name, a set of inputs, a set of outputs, and an executionBody. Both inputs and outputs are connected to slots at run time and hold messages; the execution body is a piece of Java code that implements the activities that must be carried out. Inside the execution body, a software engineer may have access to the messages held in the inputs and outputs.

Every task must fulfill the following invariants:

```
context Task
  inv: inputs->union(outputs)->isUnique(n: Name | n)
  inv: inputSlots->collect(s: Slot | s.relatedInput) = inputs
  inv: outputSlots->collect(s: Slot | s.relatedOutput) = outputs
```

These invariants state that both inputs and outputs must have unique names, that no input or output can be disconnected from a slot, and that no input or output is connected to more than one slot. Note that every slot has a property called `relatedInput` and a property called `relatedOutput`; they indicate to which task inputs and outputs they are connected, respectively. Thus, our invariants require that the set of related inputs of the input slots must coincide with the set of inputs of every task; similarly, the set of related outputs of the output slots must coincide with the set of outputs of every task. Together, these invariants guarantee that every input or output is connected to one and only one slot.

Every slot must fulfill the following invariants:

```
context Slot
  inv: not (target = source)
  inv: target.inputs->includes(relatedInput)
  inv: source.outputs->includes(relatedOutput)
  inv: let sourceProcess: Process = Process.allInstances()->
        any(p: Process | p.tasks->union(p.entryPorts.tasks)->
```

```

    union(p.exitPorts.tasks)->includes(self.source)) in
  let targetProcess: Process = Process.allInstances()->
    any(p: Process | p.tasks->union(p.entryPorts.tasks)->
      union(p.exitPorts.tasks)->includes(self.target)) in
  sourceProcess = targetProcess

```

These invariants state that every slot must connect different tasks, that they must be properly connected to the inputs and outputs of the corresponding tasks, and that they cannot connect tasks in different processes. Note that the association between `Process` and `Slot` is not backwards navigable because of the problem to model interslots; this implies that we need to calculate explicitly the process to which the source and the target tasks of every slot belong. To calculate it, we need to iterate over the whole set of process instances to find a process whose tasks, union the tasks of its entry and exit ports contain the source or the target task of every slot.

8.2.5 Datatypes

In the metamodel, we refer to a number of datatypes, namely:

Name: This type represents a subset of Java identifiers. It is represented as a `String` that satisfies the following regular expression: `[a-zA-Z_]([a-zA-Z_0-9])*`.

HostName: This type represents a subset of DNS host names. It is represented as a `String` that satisfies the following regular expression: `[a-zA-Z0-9\-\]{1,62}((\.[a-zA-Z0-9\-\]{1,62})+\.[a-zA-Z]{2,6})?`.

JndiName: This type represents a subset of JNDI names. It is represented as a `String` that satisfies the following regular expression: `[a-zA-Z_@$]([a-zA-Z_@$0-9])*(/[a-zA-Z_@$0-9]+)*`.

PositiveInteger: This type represents 16-bit positive integers. It is represented as an `int` within the following range: `minInclusive = 0` and `maxInclusive = 65534`.

They allow us to constraint some properties of the metamodel that need to be copied verbatim by our transformations.









Icon	Class	Icon	Class
	Application		IntegrationLink
	Process		ApplicationLink
	EntryPort		Slot
	ExitPort		Task

Table 8.1: Concrete syntax.

8.3 Concrete syntax

Table §8.1 shows the concrete syntax we use to represent the classes provided by our abstract syntax. Since tasks are provided in toolkits that are not part of the core language, the symbol that we depict in Table §8.1 to represent them is generic. (cf. Section §8.4 for a complete description of our general-purpose toolkit.) Note the small rounded connectors on the sides of the icon; they represent the inputs and the outputs. Slots are connected to tasks using these rounded connectors. Note that the syntax regarding processes and ports is abbreviated, *i.e.*, this is the syntax used to hide the details; they both are containers, which implies that they can be represented making it explicit their internal structure, as well.

8.4 General-purpose toolkit

In the previous sections, we have dealt with tasks in an abstract manner. In this section, we provide an insight into a general-purpose toolkit that accompanies Guaraná DSL. Figure §8.4 sketches the abstract classes that help classify concrete tasks according to their intended semantics, namely: routers, which do not change the state of the messages they process, but route them through a process, cf. Table §8.2; modifiers, which help add or remove data from messages, but do not alter their schemata, cf. Table §8.3; transformers, which help transform one or more messages into a new message with a different schema, cf. Table §8.4; stream dealers, which allow to compress, cipher, or encode messages, cf. Table §8.5; mappers, which change the format of the messages they process, *e.g.*, from a stream of bytes into an XML document,

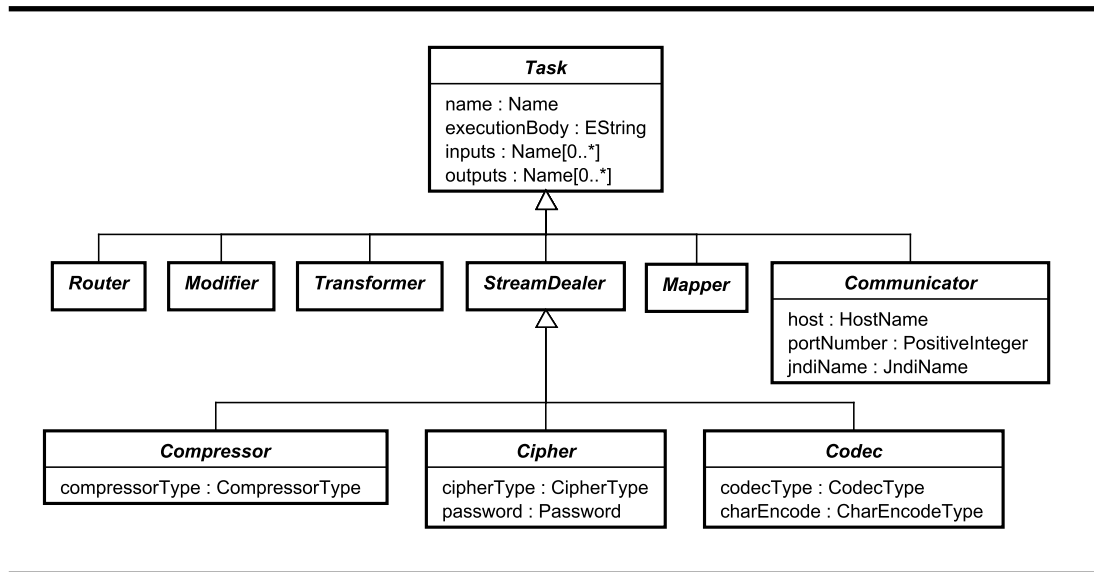


Figure 8.4: Partial view of Guaraná DSL's general-purpose toolkit.

cf. Table §8.6; and communicators, which are used to interact with binding components, cf. Table §8.7.

The general-purpose toolkit supports several integration patterns [54]. Whereas most of them are supported by a single task, others are supported by means of the composition of tasks or by means of the configuration of processes, ports, and tasks, cf. Table §8.8 and §8.9, respectively.

8.5 Summary

In this chapter, we have described the abstract and concrete syntax of our Domain-Specific Language to design Enterprise Application Integration solutions at a high-level of abstraction. Furthermore, we have introduced a ready-to-use general-purpose toolkit that supports the most common integration patterns to design integration solutions.

Notation	Task	Description
	Correlator	Analyses inbound messages and outputs sets of correlated ones.
	Merger	Merges messages from different input slots into one output slot.
	Resequencer	Reorders messages into sequences with a pre-established order.
	Filter	Filters out unwanted messages.
	IdempotentTransfer	Removes duplicated messages.
	Dispatcher	Dispatches a message to exactly one slot.
	Distributor	Distributes messages to one or more slots.
	Replicator	Replicates a message to all of the output slots.
	SemanticValidator	Validates the semantics of a message.
	CustomRouter	Allows for routing a message according to custom semantics.

Table 8.2: Router tasks.










Notation	Task	Description
	Slimmer	Removes contents from the body of a message according to a static policy.
	ContextBasedSlimmer	Removes contents from the body of a base message according to a dynamic policy that is provided by a context message.
	ContentEnricher	Adds static contents to the body of a message.
	ContextBasedContentEnricher	Adds dynamic contents from a context message to the body of a base message.
	HeaderEnricher	Adds static contents to the header of a message.
	ContextBasedHeaderEnricher	Adds dynamic contents from a context message to the header of a base message.
	HeaderPromoter	Promotes a part of the body of a message to its header.
	HeaderDemoter	Demotes a part of the header of a message to its body.
	CustomModifier	Allows to modify the header and body of a message according to custom semantics.

Table 8.3: Modifier tasks.

Notation	Task	Description
	Translator	Transforms the body of a message from one schema into another.
	Splitter	Splits a message that contains repeating elements into several messages.
	Aggregator	Constructs a new message from several messages produced previously by a Splitter.
	Chopper	Breaks a message into two or more messages.
	Assembler	Constructs a new message from two or more messages.
	CrossBuilder	Constructs a new message that contains the cartesian product of all inbound messages.
	CustomTransformer	Allows for transformation of a message according to custom semantics.

Table 8.4: Transformer tasks.

Notation	Task	Description
	Zipper	Compresses a message.
	Unzipper	Decompresses a message.
	Encrypter	Encrypts a message.
	Decrypter	Decrypts a message.
	Encoder	Encodes a message.
	Decoder	Decodes a message.

Table 8.5: Stream dealer tasks.

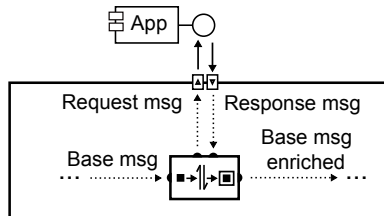
Notation	Task	Description
	Stream2XML	Maps a stream of bytes onto an XML message.
	XML2Stream	Maps an XML message onto a stream of bytes.

Table 8.6: Mapper tasks.

Notation	Task	Description
	InCommunicator	Used in ports to read messages.
	OutCommunicator	Used in ports to write messages.

Table 8.7: Communicator tasks.

Composite enquirer



Equivalent pattern

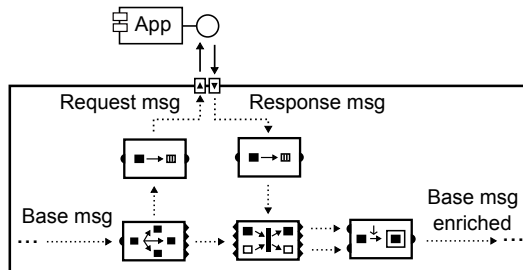


Figure 8.5: The Enquirer pattern.

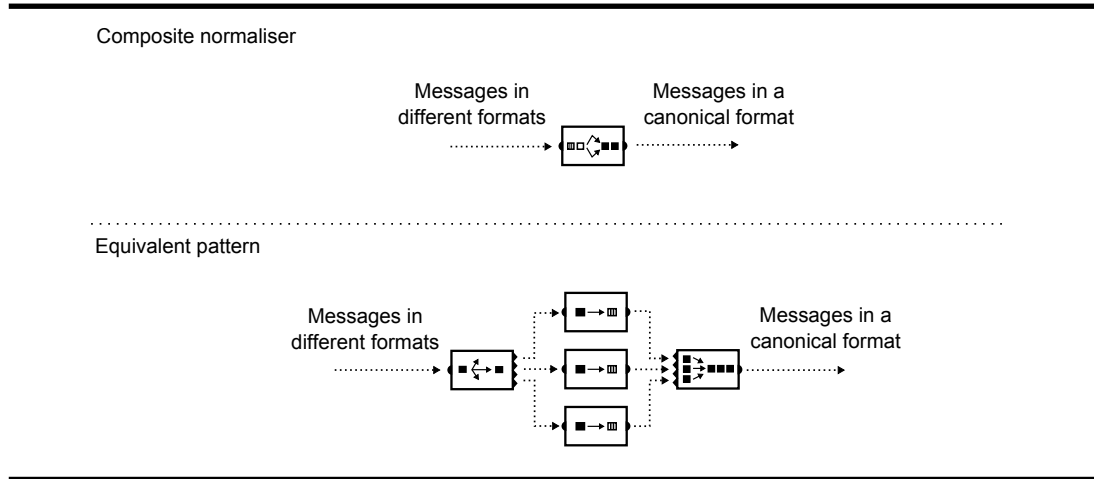


Figure 8.6: The Normaliser pattern.

Notation	Task	Description
	Enquirer	Takes a base message and uses a pair of request-response messages to enrich it with the contents from the response message. This pattern appears frequently when soliciting information from an application. Figure §8.5 shows how it is implemented.
	Normaliser	Takes messages in different formats and uses a set of translator tasks to transform them into a canonical format. Figure §8.6 shows how it is implemented.

Table 8.8: Composite tasks.

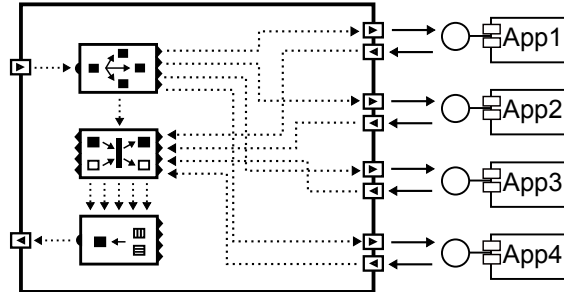


Figure 8.7: The Scatter-Gather pattern.

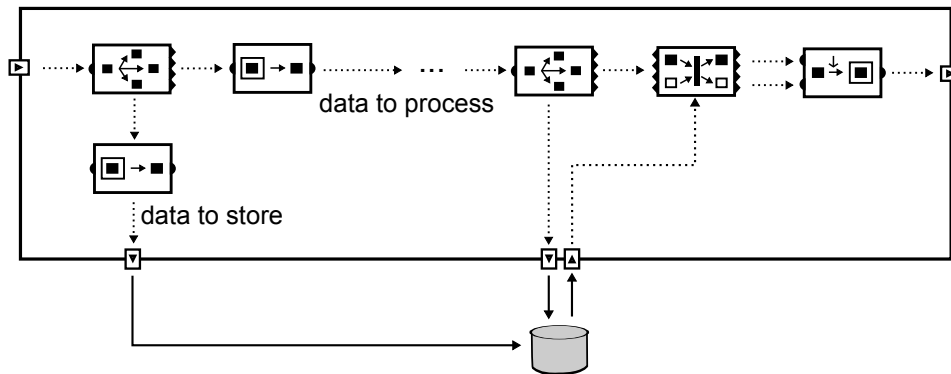


Figure 8.8: The Claim Check pattern.

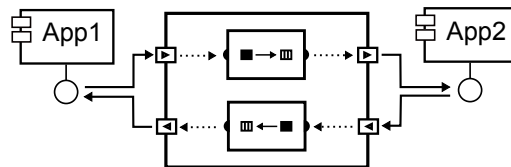


Figure 8.9: The Message Bridge pattern.

Pattern	Description
Scatter-Gather	Gathers information from multiple assets using the same request message. All responses are used to assemble a unique message that is returned as response message to the original request. Figure §8.7 shows how it is implemented.
Claim Check	Temporarily reduces the contents of a message that goes through an integration workflow. This pattern uses an external persistent store. Figure §8.8 shows how it is implemented.
Message Bridge	Synchronises two different assets so that messages available on one of them are also available on the other. Figure §8.9 shows how it is implemented.

Table 8.9: *Configuration patterns.*

Chapter 9

Software Development Kit

*Design is not just what it looks like and feels like.
Design is how it works.*

Steve P. Jobs, American businessman & inventor (1955-2011)

This chapter is devoted to present our Software Development Kit that supports software engineers to implement integration solutions. Section §9.1 provides an introduction to the Software Development Kit. Section §9.2 describes the layer in our Software Development Kit that implements the abstractions in our Domain-Specific Language. Section §9.3 presents our general-purpose toolkit, which provides a ready-to-use implementations of building blocks, such as tasks and adapters. Finally, Section §9.4 summarises the chapter.

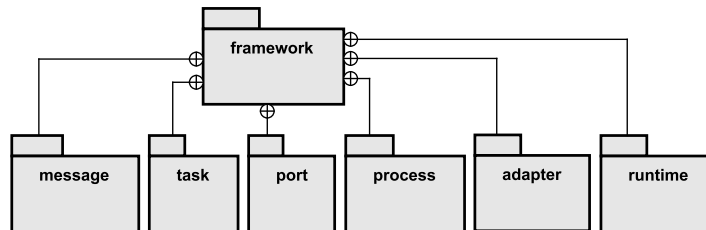


Figure 9.1: Packages of which our framework is composed.

9.1 Introduction

This chapter describes our Software Development Kit, to which we refer to as Guaraná SDK. Guaraná SDK is the Java-based software tool that we provide to implement Enterprise Application Integration solutions based on integration patterns. It was designed to be used by means of a command-query API [31]. This tool is composed of two layers, namely: framework and toolkit. The former provides a number of classes and interfaces that implement the abstractions of our Domain-Specific Language, and the latter extends some abstractions in the framework to provide a general-purpose toolkit, ready-to-use implementations of building blocks, such as tasks and adapters. By framework we mean a layer that provides abstract and concrete implementations and is able to execute them.

9.2 The framework layer

In this section, we describe the framework layer. Figure §9.1 provides an overview of this layer by showing the six packages of which it is composed. In the following subsections we describe each package.

9.2.1 Messages

Messages are used to wrap the data that is manipulated in an integration solution. They are composed of a header, a body and one or more attachments, cf. Figure §9.2.

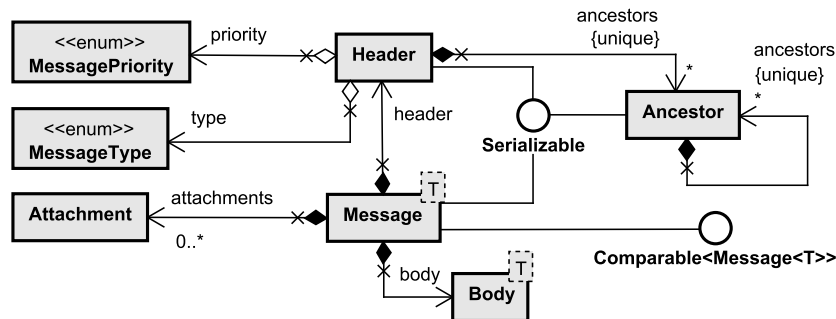


Figure 9.2: Message model.

The header includes custom properties and the following pre-defined properties (not shown in Figure §9.2): message identifier, correlation identifier, sequence size, sequence number, return address, expiration date, message priority, message type, and list of ancestors. The message identifier is represented using an immutable universally unique identifier value of 128-bits, which is automatically assigned to every message when they are created. The correlation identifier holds the identifier of another message to which the current message is correlated. Sequence size and sequence number are used to identify a message in a sequence of messages so that they can be grouped. The expiration date allows to set a deadline after which a message is considered outdated for further processing. The message priority is an enumerated value, namely: lowest, low, normal (default), high, and highest. The message type is an enumerated value that indicates whether the message represents a command, an event (default), a request, or a response. A command message aims to invoke an operation at its destination without expecting any responses; an event message is used for asynchronous notification purposes and carries data that keeps applications up to date; a request message is similar to a command message, however it always expects a reply that is a response message. The list of ancestors allows to track which messages originate from which ones; this is important in order to find out which messages have been processed as a whole, and from a so-called correlation.

The body holds the payload data, and its type is defined by the template parameter in the message class. Attachments allow messages to carry extra pieces of data associated with the payload, e.g., an image or an e-mail message. Data in the attachments are not intended to be processed, which is not

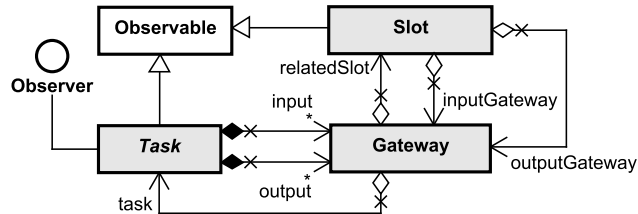


Figure 9.3: Task model.

a shortcoming at all; bear in mind that messages are defined by the users, so they can freely decide which information is stored in the body and which information is carried forward as attachments. Simply put, an attachment is a piece of data that is required to build the resulting messages of an integration solution, but not necessary during its processing.

Messages implement two interfaces so that they can be serialised and compared, respectively. Serialisation is required to deep copy, to persist, and to transfer messages; comparison enables the integration solution to process them according to their priority.

9.2.2 Tasks

This package provides the foundations to implement domain-specific tasks in specialised toolkits, *cf.* Figure §9.3. Roughly speaking, a task models how a set of inbound messages must be processed to produce a set of outbound messages, *e.g.*, routing the inbound messages, modifying them, transforming them, performing time-related actions, stream-oriented actions, mapping them to/from objects, or reading and writing messages, to name a few categories that are supported by the toolkit introduced in Section §9.3.

Tasks communicate indirectly by means of slots to which they have access by means of so-called gateways. A slot is an in-memory priority buffer that helps transfer messages asynchronously so that no task has to wait until the next one is ready to start working. Gateways act like a connection point between a slot and a task, by providing an interface to add/take messages to/from slots.

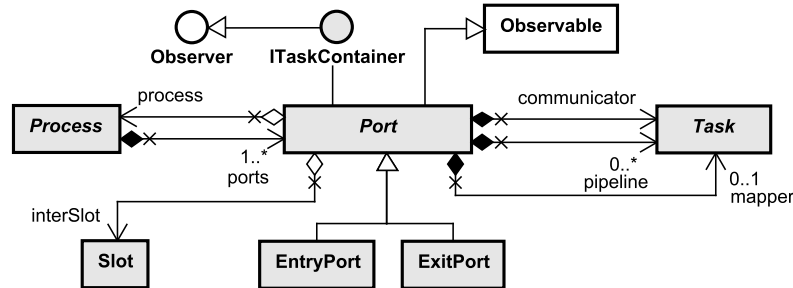


Figure 9.4: *Port model.*

Tasks become ready to be executed according to a time criterion or a slot criterion. In the former case, a task becomes ready to be executed periodically, after a user-defined period of time elapses since it became ready for the last time; in the later case, it becomes ready every time there is a new message available in every input slot. Note that becoming ready for execution just implies that the task is flagged so that the Runtime System can assign a thread to execute it; this does not entail that the task produces a set of outbound messages, but that it can examine its input slots and perform an action if the appropriate messages are found. For instance, a merger is a task that reads messages from two or more slots and merges them into one slot; this task can transfer messages as they are available. Contrarily, a context-based content enricher is a task that reads a base message and a context message from two slots and uses the later to enrich the former; note that this task becomes ready to perform its enrichment action when the base and the context messages are simultaneously available.

Both slots and tasks are observable objects, which means that they can notify other objects of changes to their state; in addition, tasks are observer objects since they monitor slots.

9.2.3 Ports

Ports abstract processes away from the communication mechanism in an inter-process communication or in the communication of the integration solution with an application, cf. Figure §9.4.

Note that every port must be associated with a process, and that we distinguish between entry and exit ports. The former are ports that allow to read messages from an application or another process; the latter are ports that allow to write a message to a process or an application.

Internally, ports are composed of tasks and one of them must be a communicator. Communicators are the tasks that allow to actually read or write a message, namely: in communicators are used to read a message in raw form a process or application; contrarily, out communicators are used to write a message in raw form to a process or an application. By raw form, we mean a stream of bytes that is understood by the corresponding process or an application. Inside ports, communicators interact with a pipeline of stream-oriented tasks, which also includes a mapper task. An in communicator passes every message read on to the pipeline; contrarily, an out communicator receives messages from the pipeline to write them. The pipeline is used as a pre/post-processor that decrypts/encrypts, decodes/encodes, or unzips/zips this stream of bytes. The pipeline in an entry port ends with a mapper task that transforms the resulting stream of bytes into a message; the pipeline in an exit port begins with a mapper that transforms a message into a stream of bytes.

Note that ports also have a so-called inter-slot. We use this term to refer to the slots that allow the last task in an entry port to send messages to the first task in a process or the last task in a process to send messages to the first task in a port.

The `ITaskContainer` interface defines an interface every container of tasks must implement. It basically allows to add, remove, get, search, and count tasks. In addition, this interface extends the `Observer` Java interface so that a container centralises notifications received from its internal tasks. This feature is important because containers can then be notified about tasks that are ready to be executed. Not only implement ports the `ITaskContainer`, but they are also observable elements, *i.e.*, they can both observe and produce notifications.

9.2.4 Processes

Processes are the central processing units in an integration solution, *cf.* Figure §9.5. They are composed of ports and tasks, implement interface `ITaskContainer`, and extend class `Observable`. The reason why processes are observable is that they are just an abstraction that helps organise

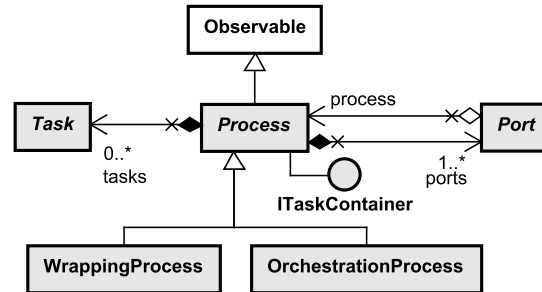


Figure 9.5: *Process model.*

groups of tasks that co-operate to achieve a goal; from the point of view of the Guaraná SDK, they are just a container that reports which of their tasks are ready for execution to an external observer. A process may have several observers, *e.g.*, to log or to monitor its activities; however, the most important one is a Runtime System, which we describe in the following section.

Processes serve two purposes, namely: there are processes that allow to wrap applications and processes that allow to orchestrate a workflow. The former are reusable processes that endow an application with a message-oriented API that simplifies interacting with it. Implementing such a wrapping process may range from using a JDBC driver to interact with a database to implementing a scrapper that emulates the behaviour of a person who interacts with a user interface. Orchestration processes, on the contrary, are intended to orchestrate the interactions with a number of services, wrapping processes, and other orchestration processes. Independently from their role, processes are composed of ports and tasks.

9.2.5 Adapters

This package provides the foundations to implement adapters in specialised toolkits, *cf.* Figure §9.6. Adapters are the piece of software that implements the low-level communication protocol that is necessary to interact with the processes or applications involved in an integration solution. The framework layer provides two interfaces to describe the operations used by ports to read and write messages.

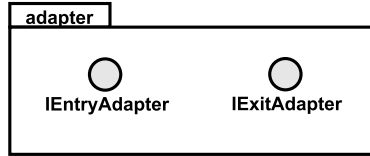


Figure 9.6: Adapter model.

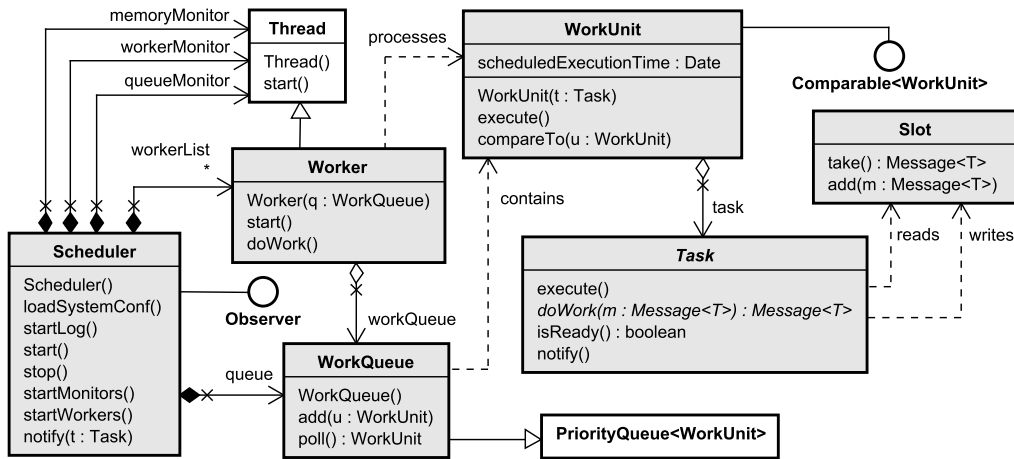


Figure 9.7: Task-based runtime model.

9.2.6 The Runtime System

The model of our Runtime System is presented in Figure 9.7. Scheduler is the central class since its objects are responsible for coordinating all of the activities in an instance of our Runtime System. Note that this class is not a singleton since we do not preclude the possibility of running several instances concurrently. At runtime, a scheduler owns a work queue, a list of workers, and three monitors.

The work queue is a priority queue that stores work units to be processed. A work unit has a reference to a task and a scheduled execution time before which it cannot execute. Note that class Task is abstract, which means that our Runtime System is not bound with a particular set of tasks; this allows

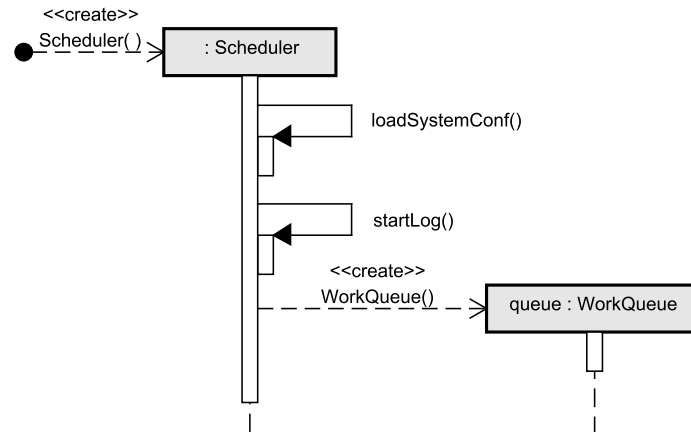


Figure 9.8: *Initialising the Runtime System.*

to create specific-purpose task toolkits that can be plugged into the Runtime System. Usually, the scheduled execution time of a work unit is set to the current time, which means that the corresponding task can execute as soon as possible; if it is set to a time in future, then the corresponding task is delayed until that time has elapsed. This is very useful to implement tasks that need to execute periodically, e.g., a communicator that polls an application every minute.

Class *Worker* extends the standard *Thread* class, *i.e.*, objects of this class run autonomously. Each worker is given a reference to the work queue, from which they concurrently poll work units to process.

The monitors gather statistics about the usage of the memory, the CPU, and the work queue. The memory monitor registers information about both heap and non-heap memory; the worker monitor registers the user- and the system-time worker objects have consumed; and, the queue monitor registers the size of the queue and the total number of work units that have been processed. Monitors were implemented as independent threads that run at regular intervals, gather the previous information, store it in a file, and become idle as soon as possible.

Schedulers are configured using a simple XML file with information about the number of workers, the files to which the monitors dump statistics, the frequency at which they must run, and the logging system used to report warn-

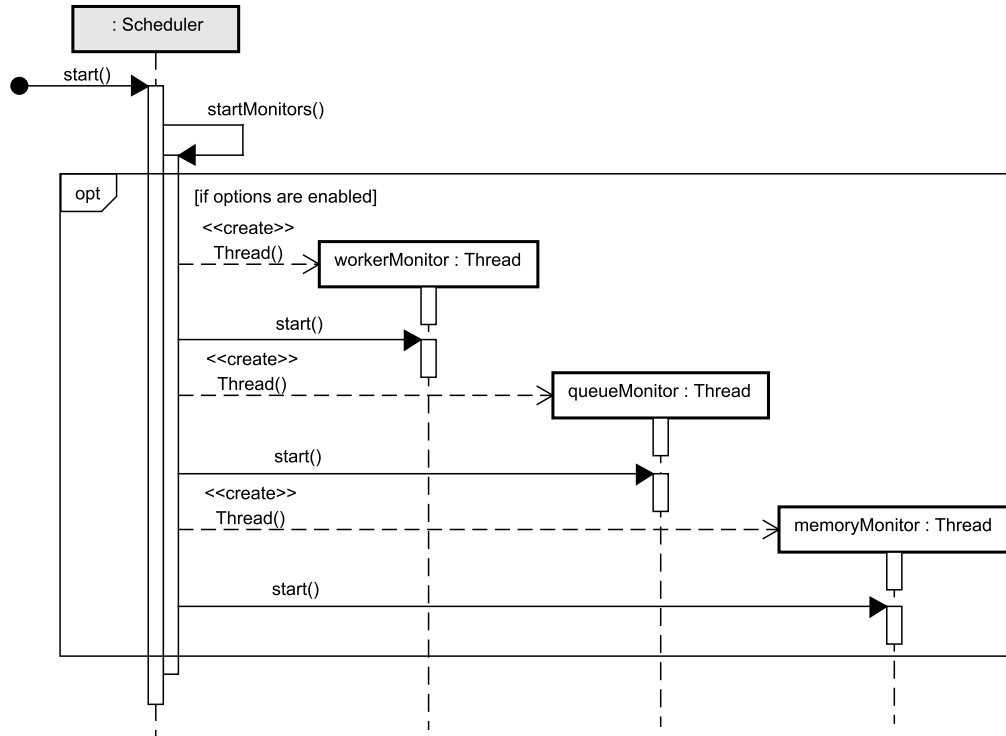


Figure 9.9: *Creating and starting monitors.*

ings and errors. Figure §9.8 shows the sequence of operations involved in the initialisation of a scheduler. The first operation loads the configuration file and analyses it; then, the logging system is started, and a work queue is created.

Note that engines are not started when they are created. It is the user who must decide when to start them using the `start` operation. This operation causes the invocation of two other operations, namely: `startMonitors` and `startWorkers`. The former starts the monitors that have been activated in the configuration file, cf. Figure §9.9, and the later creates and starts the workers.

Figure §9.10 shows the sequence of operations required to create and start the workers. Note that they are started asynchronously by invoking operation `start`. The business logic of a worker is defined inside its `doWork` operation. This operation implements a loop that enables the workers to poll the work queue as long as the scheduler is not stopped. When a work unit is polled, the worker first checks its scheduled execution time; if it has expired, then the

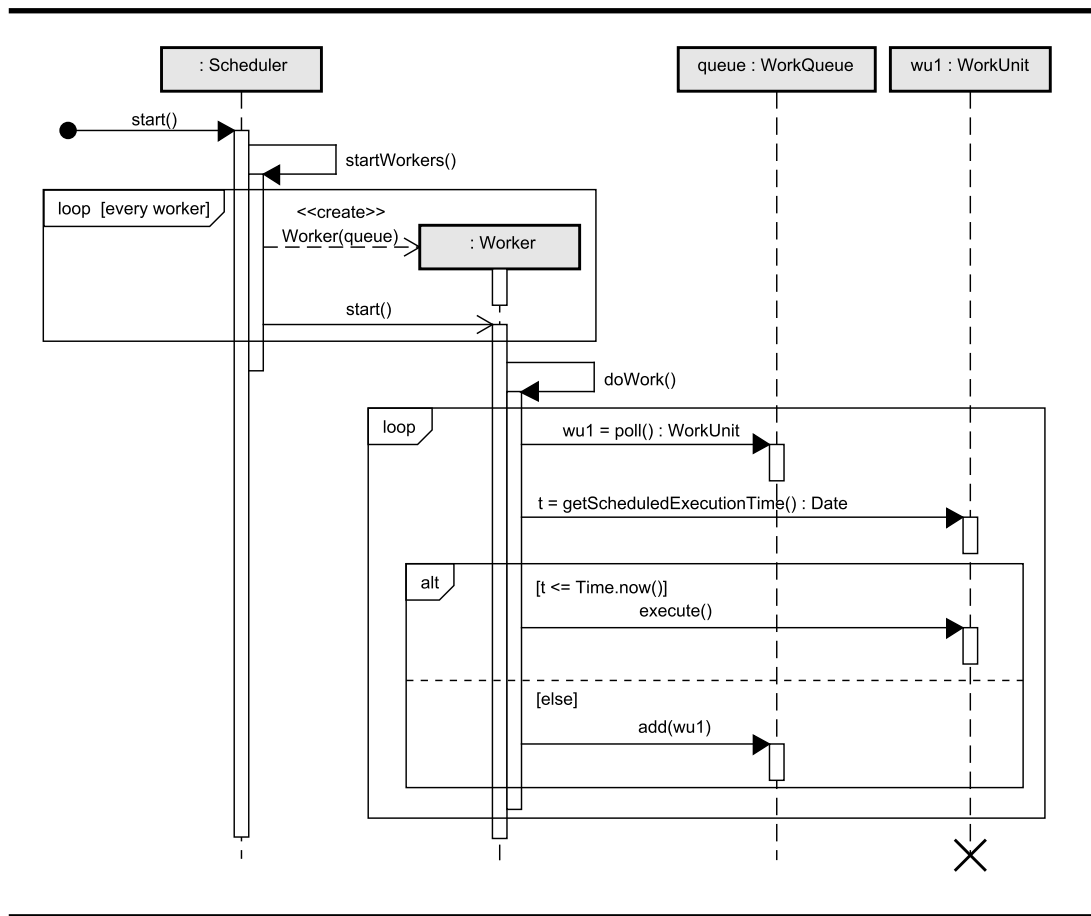


Figure 9.10: *Creating and starting workers.*

task can be executed immediately; otherwise, the work unit is delayed until the deadline expires. Note that this strategy allows workers to keep working as long as there is a task ready to be executed.

Processing a work unit requires invoking operation `execute` on the associated task, which first packages the input messages and then invokes operation `doWork`, which depends completely on the task toolkit being used. Then, the task writes its output messages to the appropriate slot, which in turn notifies the tasks that read from them. These tasks then determine if they become ready for execution or not; in the former case, the tasks notify the container to which they belong. Containers of tasks propagate every notification they receive to the scheduler. For every task notification that the scheduler receives, it creates a new work unit and appends it to the work queue, cf. Figure §9.11.

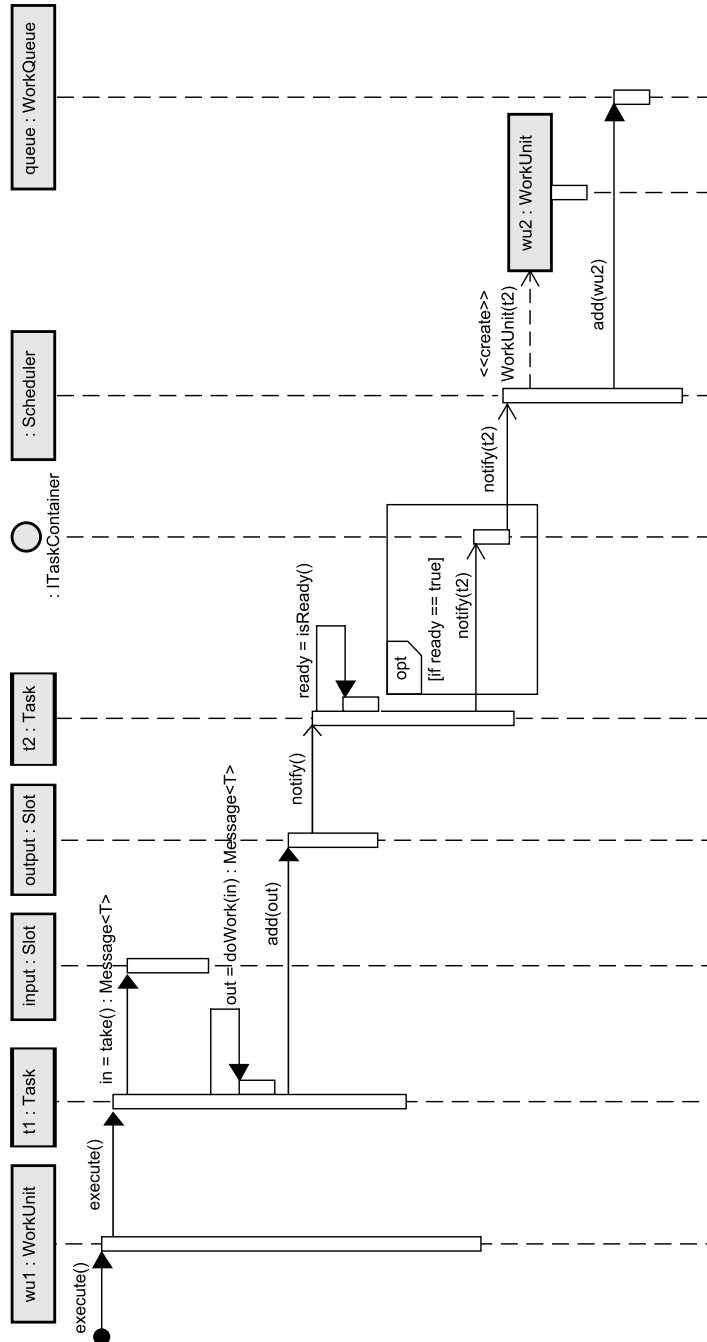


Figure 9.11: Executing a WorkUnit.

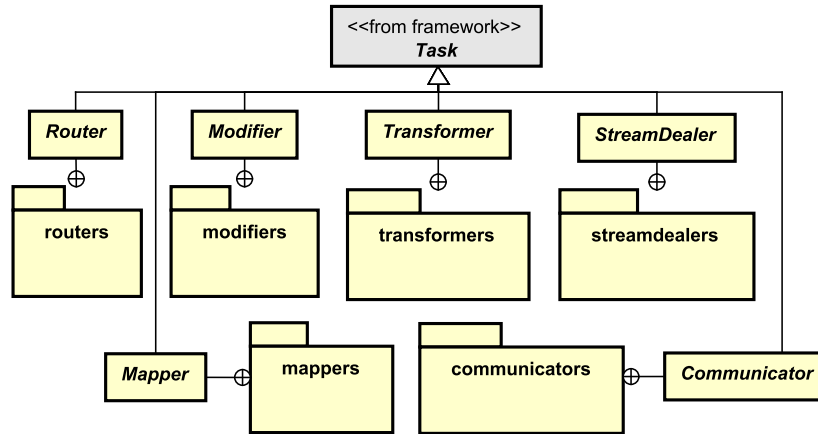


Figure 9.12: Task model in the toolkit.

9.3 The general-purpose toolkit layer

The framework provides two extension points, namely: Task and Adapter. We have designed a core toolkit that provides extensions to deal with a variety of tasks that support the majority of integration patterns in the literature [54], and provide active and passive adapters that enable the use of several low-level communication protocols.

This toolkit provides extensions to the Task class, cf. Figure §9.12. In the following descriptions we use term schema to refer to the logical structure of the body of a message. It may range from a DTD or an XML schema to a Java class. The first level of extension is composed of additional abstract classes that are intended to make it explicit several categories of integration patterns, namely:

Router: a router is a task that does not change the messages it processes at all, but routes them through a process. This includes filtering out messages that do not satisfy a condition or replicating a message, to mention a few tasks in this category.

Modifier: a modifier is a task that adds data to a message or removes data from it as long as this does not result in a message with a different schema. This includes enriching a message with contextual information

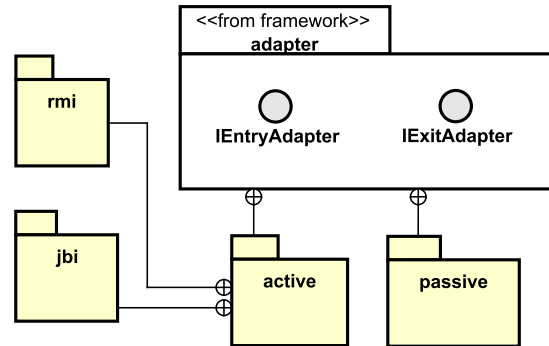


Figure 9.13: Adapter model in the toolkit.

or promoting some data to its headers, to mention a few examples in this category.

Transformer: a transformer is a task that translates one or more messages into a new message with a different schema. Examples of these tasks include splitting a message into several ones or aggregating them back.

StreamDealer: a stream dealer is a task that deals with a stream of bytes and helps zip/unzip, encrypt/decrypt, or encode/decode it.

Mapper: a mapper is a task that changes the representation of the messages it processes, e.g., from a stream of bytes into an XML document.

Communicator: a communicator is a task that encapsulates an adapter. Communicators serve two purposes: first, they allow adapters to be exported to a registry so that they can be accessed remotely; second, a communicator can be configured to poll periodically a process or application using an adapter.

There is a package associated with every of the previous tasks. They provide a variety of specific-purpose implementations in each integration pattern category, cf. Section §8.4.

In the previous section, we mentioned that ports use communicators to communicate with other processes or applications. As we mentioned before, they rely on adapters, which can be either active or passive, cf. Figure §9.13.

An active adapter allows to poll the process or application with which it interacts periodically; contrarily, a passive adapter aims to export an interface to a registry, so that other applications or processes can interact with it. Note that entry and exit ports can be implemented using either active or passive adapters.

The active package is divided into two packages to provide implementations that are based on the JBI and RMI protocols, respectively. Note that supporting JBI adapters allows to plug Guaraná SDK into a variety of ESBs; for example, our reference implementation is ready to be plugged into Open ESB [90]. This, in turn, allows Guaraná SDK processes to have access to a variety of applications in current software ecosystems, including files, databases, web services, RSS feeds, SMTP messaging systems, JMS queues, DCOM servers, and so on. The `rmi` package provides several implementations that are intended to be used to interact with an RMI-compliant server.

9.4 Summary

In this chapter, we have described Guaraná SDK, which is our software tool to implement Enterprise Application Integration solutions designed with our Domain-Specific Language. The architecture of this tool is organised into two layers, namely: framework and toolkit. The former provides a number of classes and interfaces that implement the abstractions of our Domain-Specific Language, whereas the latter extends some abstractions in the former to provide concrete adapters and tasks that support several integration patterns.

Chapter 10

Model-to-text Transformations

*The universe is transformation.
Marcus Aurelius, Roman Emperor (121-180)*

Our aim in this chapter is to report on the transformations we have devised to translate models that were described with our Domain-Specific Language into executable Java code that uses our Software Development Kit. Section §10.1 introduces our work and the language used to devise the transformations. Section §10.2 describes how we transform processes. Section §10.3 presents how ports are transformed. Section §10.4 introduces the transformation of tasks. Section §10.5 describes how communicators are transformed. Section §10.6 describes how to generate a starter Java class that allows to run the integration solution in our Runtime System. Finally, Section §10.7 summarises the chapter.

10.1 Introduction

In this chapter, we report on the transformations we have devised to translate models described with our Domain-Specific Language into executable Java code that uses our Software Development Kit. We have devised model-to-text transformations using the MOFScript language [84]. The resulting code shall be executed in our Runtime System. The reason for using MOFScript is that our research group had already experience on working with this language.

Unfortunately, the original transformations are very verbose, which makes them not appropriate to be shown here. In the sequel, we have resorted to a simplified notation in which the executable code is enclosed within angle brackets; the remaining text is assumed to be copied verbatim.

10.2 Transforming processes

This transformation is executed on every process, and it produces a Java class that includes slots, tasks, entry and exit ports declarations, plus a constructor that initialises them all. The transformation is as follows:

```
1: package <Process.name>;
2: <Import classes>
3: public class <Process.name> extends Process {
4:     <Slots declaration>
5:     <Tasks declaration>
6:     <EntryPorts declaration>
7:     <ExitPorts declaration>
8:
9:     public <Process.name>() {
10:         <Slots initialisation>
11:         <EntryPorts initialisation>
12:         <ExitPorts initialisation>
13:         <Tasks initialisation>
14:     }
15: }
```

Line [S1](#) declares a package with the name of the process being transformed; the classes that correspond to the ports shall also be placed within this package

to avoid name clashes with other ports in other processes.

At line §2, the transformation constructs the import statements required to have access to the classes the Runtime System provides. Line §3 declares the class for the process, which extends the Process class provided by the Runtime System. Inside this class, lines §4–§7 introduce a number of attributes that shall reference the slots, tasks, and ports of which the process being transformed is composed. Lines §9–§14 provide a constructor.

Note that none of the previous declarations or initialisations are difficult, since they just need to iterate over the appropriate properties of a model and output Java declarations or initialisations. The only part that requires a little more explanation is the transformation to initialise ports and tasks. Here we report on the former; the latter is complex enough to deserve a new section.

What follows is the transformation to initialise the entry ports:

```
1: <Process.entryPorts->forEach(p: EntryPort) {
2:     <p.name> = new <p.name>();
3:     addPort(<p.name>);
4: }
```

The loop at line §1 iterates over the collection of entry ports of a process. At line §2, it outputs a new statement to create a port; recall that every port results in a class with the same name. The following line binds the port and the process to which it belongs by means of the corresponding interslot. The transformation of exit ports is equivalent.

10.3 Transforming ports

This transformation is executed on each port independently, and it results in a Java class that includes slot and task declarations, plus a constructor that initialises them all. The transformation is as follows:

```
1: package <Process.name>;
2: <Import classes>
3: public class <Port.name> extends <Port.getTypeName()> {
4:     <Slots declaration>
5:     <Tasks declaration>
6:
```

```

7:     public <Port.name>() {
8:         <Slots initialisation>
9:         <Communicator initialisation>
10:        <Tasks initialisation>
11:    }
12: }

```

Lines §1 and §2 introduce the package declaration and import the classes that are required. In the class declaration at line §3, the operation `getTypeName` is used to discover the parent class. Lines §4 and §5 declare an attribute per slot and task, respectively. Lines §8–§10 inside the constructor deal with the initialisation of the previous declarations. Recall that every port must have a communicator so that it can interact with the corresponding binding component. These tasks are dealt with by the Runtime System like any other task; however, they must be initialised in a way that deviates from the rest. We report on how to initialise tasks and communicators in following sections.

10.4 Transforming tasks

We have grouped tasks into five groups. The following transformation illustrates how to initialise tasks that have several inputs and several outputs. The rest of the groups, except for communicators and a few other tasks, are special cases of this general case.

```

1: <Task.name> = new <Task.getTypeName()>
2:   (<Task.name>, <Task.inputs.size()>, <Task.outputs.size()>) {
3:   @Override
4:   public void doWork(Exchange e) {
5:       <Task.executionBody>
6:   }
7: };
8: <Bind input slots>
9: <Bind output slots>
10: addTask(<Task.name>);

```

Note that we create anonymous classes to initialise tasks. Each concrete task is derived from a class that is provided by a toolkit, which is, in turn, discovered by means of a call to operation `getTypeName`. As a consequence, we need to override the `doWork` operation only.

The transformation at line §5 writes the execution body inside this operation. Lines §8 and §9 deal with binding the input and output slots to the corresponding inputs and outputs of this task. Finally, line §10 adds the task that has been initialised in the previous lines to the enclosing port or process.

10.5 Transforming communicators

Transforming a communicator is different from transforming other tasks because they have to interact with binding components. Next, we present the transformation that deals with InCommunicators in entry ports:

```

1: <Communicator.name> = new InCommunicator(
2:     "<Communicator.jndiName>",
3:     "<Communicator.host>",
4:     <Communicator.portNumber>);
5: <Bind output slot>
6: setCommunicator(<Communicator.name>);

```

InCommunicators are published as remote objects in an RMI registry so that they can be invoked by binding components. This is the reason why the constructor gets a JNDI name which identifies the communicator inside the registry, as well as the host name and the port number where the RMI registry is running. Line §5 binds the single slot connected with this communicator to its output. Finally, line §6 sets the communicator to the enclosing entry port.

OutCommunicators are a little more cumbersome since they need to invoke binding components to write messages. The script to transform them is as follows:

```

1: Properties props = new Properties();
2: props.setProperty("java.naming.factory.initial",
3: "com.sun.enterprise.naming.SerialInitContextFactory");
4: props.setProperty("java.naming.factory.url.pkgs",
5: "com.sun.enterprise.naming");
6: props.setProperty("java.naming.factory.state",
7: "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl"
8: );
9: props.setProperty("org.omg.CORBA.ORBInitialHost",
10: "<Communicator.host>");

```

```

11: props.setProperty("org.omg.CORBA.ORBInitialPort",
12: "<Communicator.portNumber>");
13:
14: <Communicator.name> = new OutCommunicator(
15: "<Communicator.name>",
16: new JbiExitResourceAdapter("<Communicator.jndiName>", props)
17: );
18: <Bind input slot>
19: setCommunicator(<Communicator.name>);

```

Note that one can have access to a binding component as if it were a regular EJB. This is why lines §1–§12 set up a Properties object with most of the properties required to configure a connection with an EJB. Line §14 initialises the out communicator, which requires to create a JbiExitResourceAdapter; this object implements an adapter to connect to a binding component given its JNDI name and the previous properties. Later, the out communicator is bound to its input slot in line §18 and it is registered with the enclosing exit port in line §19.

10.6 Generating the starter

The previous transformations deal with creating the classes that implement the processes and the ports of which an integration solution is composed. These are the pieces that now need to be put together by means of the following starter transformation:

```

1: package <Solution.name>;
2: <Import classes>
3: public class <Solution.name> {
4:     public static void main(String[] args) {
5:         Runtime r = new Runtime(<number of threads>);
6:
7:         <processes->forEach(p: Process) {
8:             Process <p.name> = new <p.name>();
9:             <p.name>.setTaskStateMonitor(r);
10:
11:             Collection<Port> <p.name+"Ports"> =
12:                 <p.name>.getAllPorts();
13:             for (Port pt: <p.name+"Ports">) {

```

```
14:             pt.setTaskStateMonitor(r);
15:         }
16:     ⟨⟩
17:     r.start();
18: }
19: }
```

Line §5 instantiates the Runtime System with a given number of threads. The loop in lines §7–§16 iterates over every process in the model, initialises an instance, and binds it to the Runtime System we have created previously by means of operation `setTaskStateMonitor`; furthermore, the ports are retrieved and registered with the Runtime System, as well. Registering a process or a port with a Runtime System allows it to have access to their tasks and slots and to initialise its internal data structures. Finally, the Runtime System is started at line §17. The generated Java code from all model-to-text transformations is ready to be compiled and executed on the Runtime System. Software engineers only need to configure a number of binding components, in Open ESB; they shall be used to communicate with the applications being integrated.

10.7 Summary

In this chapter, we have presented a set of model-to-text transformations that allows to automatically translate models that are described with our Domain-Specific Language into Java code that uses of our Software Development Kit. We have provided scripts for transforming processes, ports, tasks, communicators, and generating a starter Java class that allows to run the integration solution in our Runtime System.

Chapter 11

Error Detection in Integration Solutions

*It is possible to fail in many ways ...
whereas succeeding is possible only in one way.*
Aristotle, Philosopher (384 BC - 322 BC)

In this chapter, we describe our proposal to endow integration solutions with a monitor that enables the detection of errors. Section §11.1 provides an overview of the architecture of our monitoring system. Section §11.2 describes the Meta-information database shared by all of the stages of the fault-tolerance pipeline. Section §11.3 reports on the subsystem used to manage events. Section §11.4 describes the subsystem that analyses them, detects, and notifies possible errors. Section §11.5 presents the time complexity analysis of our algorithms. Section §11.6 shows the results of the experiments we have conducted considering six well-known patterns that lie at the core of most real-world integration solutions. Finally, Section §11.7 summarises the chapter.

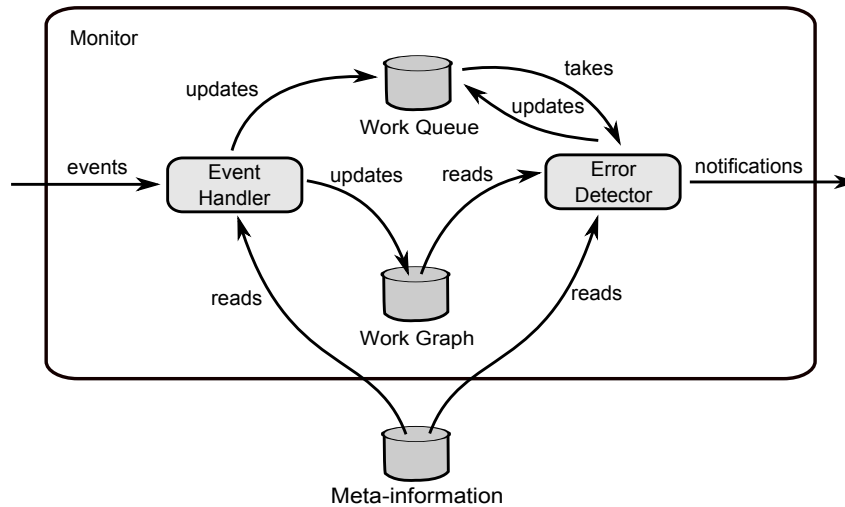


Figure 11.1: Abstract view of the monitor for error detection.

11.1 Introduction

Integration solutions are inherently distributed; they are thus vulnerable to a variety of errors that can have an impact on their normal behaviour. Errors are due to faults, which can be either permanent, *e.g.*, due to a software defect, or transient, *e.g.*, due to a resource that is temporarily unavailable. Errors that are not dealt with properly are perceived as failures by end users [2, 14]. Fault-tolerance proposals aim to help keep systems delivering their functionality in spite of faults. Typically, they can be modelled as a pipeline that goes through the following stages: event reporting, error monitoring, error diagnosing, and error recovering. The event reporting stage deals with reporting whether a port was able to handle a message or not. In the error monitoring stage, events are stored and analysed to find correlations that shall later be checked for validity. When an error is detected, a notification is created and sent to the error diagnosing stage, whose aim is to identify the cause of the error, the messages and the parties involved. The error recovering stage attempts to execute recovery actions to help the system compensate for the existence of faults and the occurrence of errors.

In this dissertation we focus on the error monitoring stage. We describe our approach to provide integration solutions with a monitor that enables the detection of errors. Our failure semantics include errors when reading from or

writing to a resource, structural, and deadline errors. Our proposal is sketched in Figure §11.1. It relies on a so-called Meta-information database that stores meta-information about the integration solutions being monitored, *e.g.*, which processes and ports are involved in an integration solution. Note that this database is external to the monitor since it aims to be shared with other stages of the fault-tolerance pipeline.

The monitor itself is composed of two subsystems and two databases, namely: Event Handler, Error Detector, Work Graph and Work Queue. The Event Handler handles events that inform about a port reading or writing a message, either successfully or unsuccessfully. It uses the events to build a graph structure that is stored in the Work Graph database; this graph keeps track of the messages processes exchange and their parent-child relationships. The Error Detector analyses this graph to find and verify correlations. A correlation is represented as a graph that has a single connected component that represents a subset of messages that are correlated to each other [55]. The Work Queue database is used as an intermediate buffer that allows the Event Handler and the Error Detector to work in total asynchrony. Every time the Event Handler processes an event, it stores a piece of information in the Work Queue database; this information instructs the Error Detector to analyse the Work Graph database at a specific point in time in order to find the correlation in which a specific message is involved. To verify correlations, the Error Detector builds on both built-in and user-defined rules; the former allows to detect communications or deadline errors; the latter allows to detect structural errors that depend on the semantics of a given process or integration solution, *i.e.*, correlations that lack messages or have more messages than expected. The only assumption we make is that the clock resolution of the monitor is enough to distinguish between every two messages that are read or written in a row; in other words, we can distinguish between multiple events that involve the same message at the same port. In practice, this is not a shortcoming since current clock resolutions are in the order of nanoseconds, whereas reading or writing to a port usually requires much more time.

It is common that integration solutions share processes, which makes them overlap or even include others. Figure §11.2 shows the design of two integration solutions that we shall use throughout this chapter to illustrate our proposal. In this example, there are two overlapping integration solutions, namely: *Solution1*, which integrates applications *App1* and *App2* by means of processes *Prc1* and *Prc2*, and *Solution2*, which integrates applications *App3*, *App4* and *App2* by means of processes *Prc3* and *Prc2*. Note that process *Prc2* is shared by both integration solutions.

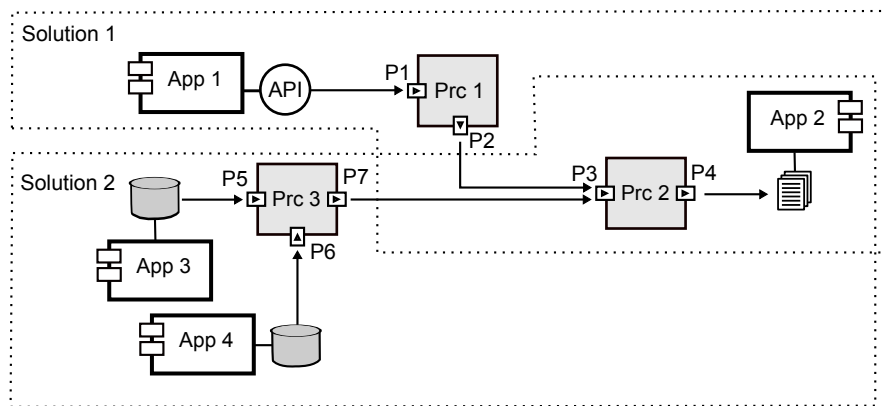


Figure 11.2: Sample integration solutions.

In the following sections we describe the subsystems and databases that comprise our approach.

11.2 The Meta-information database

In this section, we report on the meta-data we use to represent the information our proposal requires about the artefacts it monitors, *i.e.*, solutions and processes. This meta-data are stored in the Meta-information database. We do not provide details on how this database is managed since this is a typical information system with an interface that can be used by administrators to register, unregister, or list information about the integration solutions being monitored. Instead, we focus on the meta-data it stores, whose model is presented in Figure §11.3, and provide a few hints on our implementation.

Our monitoring system assumes that an integration solution is composed of at least one process, and every process must have at least two ports with different directions (`Direction::ENTRY` or `Direction::EXIT`). Every artefact, port, or rule has a name that identifies it uniquely. In addition, artefacts have a time out, which denotes the maximum time they can consume to process a set of correlated messages, and a set of rules that helps verify the correlations in which they are involved.

Rules are central to our proposal. Figure §11.4 presents the syntax we use

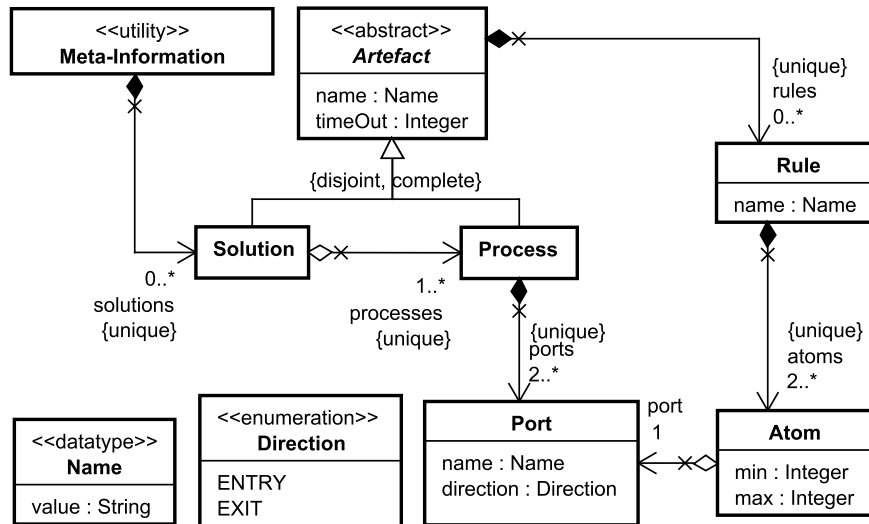


Figure 11.3: Model of the Meta-information database.

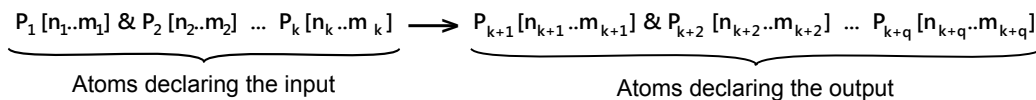


Figure 11.4: Textual syntax for error-detection rules.

- a) $P[+] = P[1 \dots \text{Integer.MAX_VALUE}]$
- b) $P[*] = P[0 \dots \text{Integer.MAX_VALUE}]$
- c) $P[?] = P[0..1]$
- d) $P[n] = P[n..n]$

Figure 11.5: Syntactic sugar for error-detection rules.

to write them textually. They are composed of two groups of atoms that are separated by an arrow. Atoms at the left hand side declare the input of the rule, whereas atoms at the right hand side declare the output of the rule, *i.e.*, if the number of messages at the left hand side atoms occurs, then it is expected that the specified number of messages in the atoms at the right hand

Prc1	Solution1
R1 = P1[1] \longrightarrow P2[?]	R5 = P1[1] \longrightarrow P4[*]
Prc2	Solution2
R2 = P3[1] \longrightarrow P4[+]	R6 = P5[1] & P6[1] \longrightarrow P4[+]
Prc3	
R3 = P5[1] & P6[1] \longrightarrow P7[2..4]	
R4 = P5[1] & P6[0] \longrightarrow P7[1]	

Figure 11.6: Sample error-detection rules.

side. Each atom is of the following form: $P[\text{min}.. \text{max}]$, where P refers to a port name, and min and max are natural numbers that represent the minimum and the maximum number of messages that are allowed at port P in a given correlation. For the sake of brevity, we use the common syntactic sugar depicted in Figure §11.5.

Figure §11.6 presents a few rules for the artefacts in the sample integration solutions we introduced in Figure §11.2. For instance, Rule R1 is associated to process `Prc1` and involves entry port `P1` and exit port `P2`; it states that a given correlation is valid if there is one message at port `P1` and zero or one correlated message at port `P2`. Similarly, Rule R6 is associated to `Solution2`; it states that a given correlation is valid if there is one message at port `P5`, one correlated message at port `P6`, and one or more correlated messages at port `P4`.

11.3 The Event Handler

Figure §11.7 depicts the model we have devised for the Event Handler. Roughly speaking, it handles events that inform about a port reading or writing a message, either successfully or unsuccessfully. The events are used to build a graph that is maintained incrementally in the `Work Graph` database. This graph records information about the messages being exchanged in an integration solution and their parent-child relationships, *i.e.*, which messages originate from which ones, for example, after executing `split` and `merge` operations. In addition, the Event Handler updates the `Work Queue` database in order to schedule the activation of Error Detector.

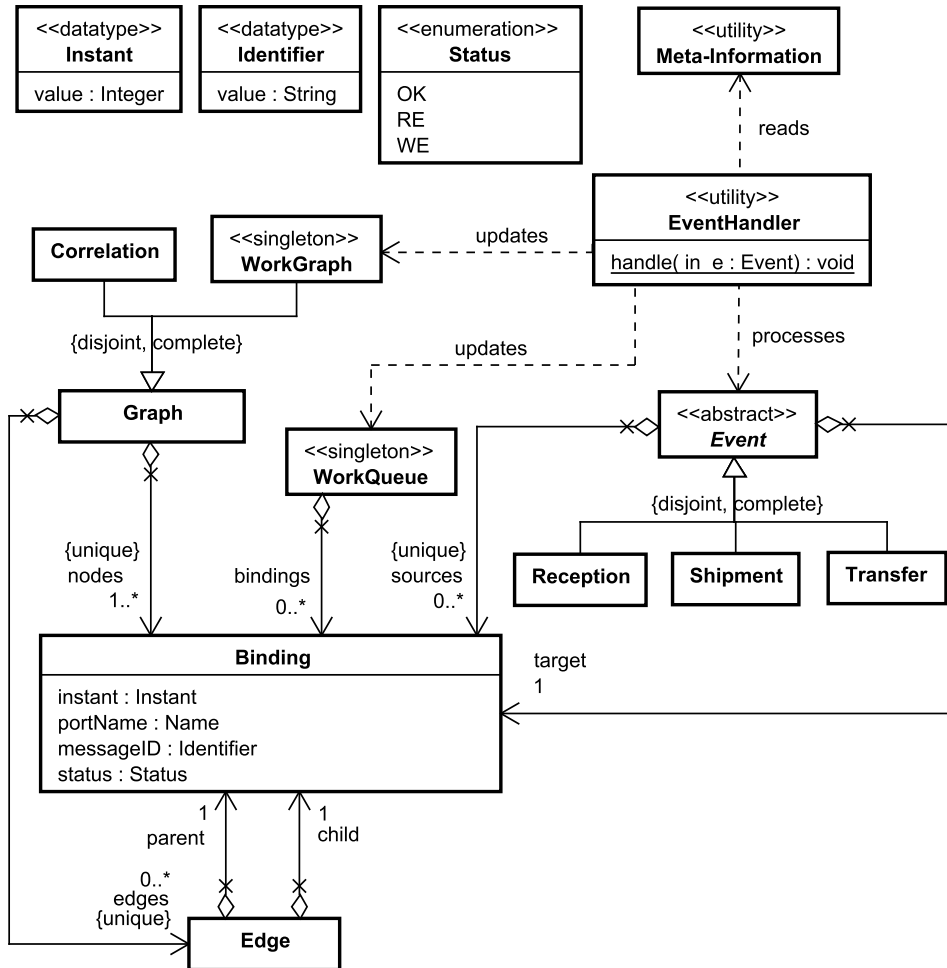


Figure 11.7: Model of the Event Handler.

Events can be of type Reception, which are notified from ports that read data from an application (either successfully or unsuccessfully) and ports that fail to read data from application or integration links, Shipment, which are notified from ports that write information (either successfully or unsuccessfully), and Transfer, which are notified from ports that succeed to read data. Every event has a target binding and zero, one, or more source bindings. We use this term to refer to the data contained in an event, namely: the instant when the event happened, the name of the port, the identifier of the message read or written, and a status, which can be either Status : :OK to mean that no problem

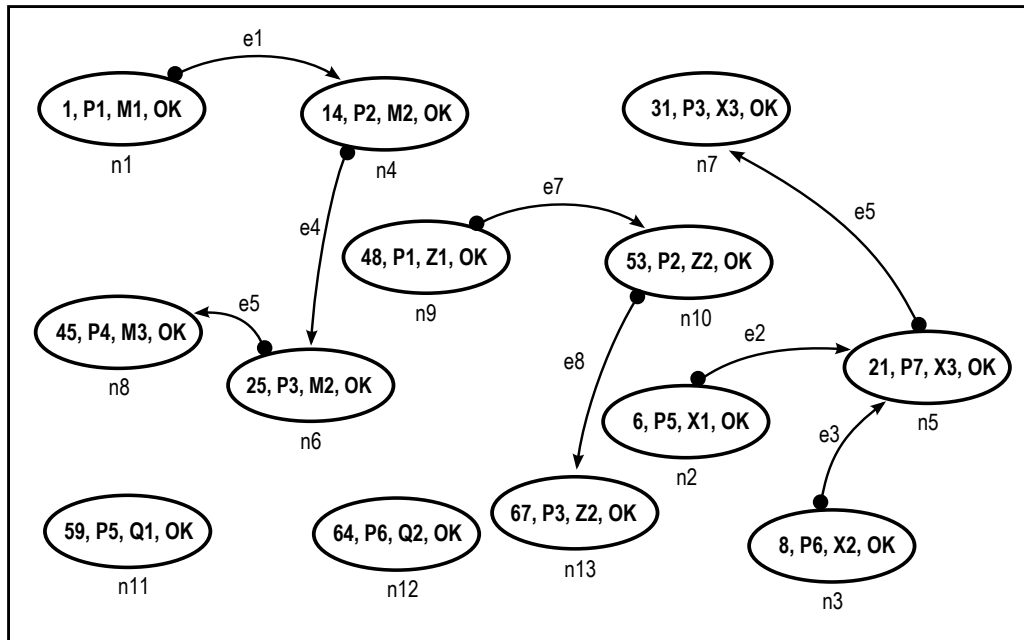
```
1: to handle(in e: Event) do
2:   p = find the process to which a port
3:   called e.target.portName belongs
4:     in the Meta-information database
5:   s = find all integration solutions to which p belongs
6:     in the Meta-information database
7:   notBefore = e.target.instant + max(s ∪ {p}).timeOut
8:   g = WorkGraph.getInstance()
9:   q = WorkQueue.getInstance()
10:  add e.target to g.nodes
11:  add e.target with priority notBefore to q
12:  for each binding b in e.sources do
13:    r = new Edge(parent = b, child = e.target)
14:    add r to g.edges
15:  end for
16: end
```

Program 11.1: *Algorithm to handle events.*

was detected, `Status::RE` to mean that there was a read error, or `Status::WE` to mean that there was a write error. Recall that the only assumption we make is that the clock resolution of the monitor is enough to distinguish between every two messages that are read or written consecutively. Thus, we assume that no confusion regarding the same message being read from or written to the same port may happen.

The `Event Handler` is implemented as a single method that handles every type of events. The algorithm for this method is presented in Program §11.1. It gets an event `e` as input and proceeds as follows: it first finds the process to which the event refers and the integration solutions to which this process belongs. Then, it computes the earliest time at which the `Error Detector` should analyse the target binding, which is the instant when the message in the target binding was read or written plus the maximum time out involved; this is a safe deadline that guarantees that every artefact should have enough time to process the corresponding correlation. Note that the time we calculate (`notBefore`) is just a hint to be interpreted as “the `Error Detector` should not analyse that binding before this time”; obviously, the sooner the binding is analysed after this time has passed, the better, but it is not a real-time require-

Work Graph



Work Queue

n1	n2	n3	n4	n5	n6	...
101	107	109	115	122	126	

Figure 11.8: Sample work graph.

ment. The algorithm then fetches both the Work Graph and the Work Queue instances and adds the target binding to them both; then, it iterates over the source bindings, if any, and adds them to the Work Graph together with an edge to link them to the target binding.

Figure §11.8 illustrates a graph that results from executing the previous algorithm on a series of bindings regarding the sample system in Figure §11.2. Ellipses denote bindings and arrows denote edges that connect parent bindings to their corresponding child bindings. For instance, n1 is the parent binding of n4, and the latter is the parent of n6, which is in turn the parent of n8. Inside each binding we represent the instant, the port name, the message id, and the status, respectively. A snapshot of the Work Queue is also presented in this figure.

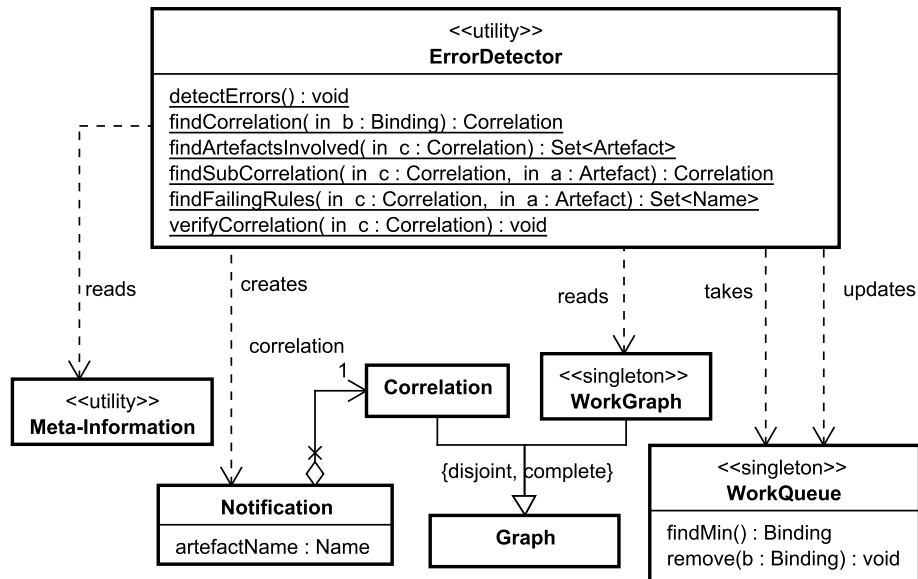


Figure 11.9: Model of Error Detector.

```

1: to detectErrors() do
2:   q = WorkQueue.getInstance()
3:   repeat
4:     b = fetch minimum of q (waiting if necessary)
5:     c = findCorrelation(b)
6:     verifyCorrelation(c)
7:     for each binding b in c.nodes do
8:       remove b from q
9:     end for
10:  end repeat
11: end

```

Program 11.2: Algorithm to detect errors.

11.4 The Error Detector

The model of the Error Detector is presented in Figure §11.9, and the algorithm to detect errors is presented in Program §11.2.

The Error Detector executes a never-ending loop in which it fetches entries from the `Work Queue`, finds the correlations in which the corresponding bindings are involved, and then verifies them to find possible errors. Recall that each binding in the `Work Queue` is scheduled to be processed not before a given time; this implies that this algorithm may need to block for some time when the earliest binding is scheduled to be analysed after the current moment. After a correlation is verified, its bindings are removed from the `Work Queue`, since analysing them again would not result in new correlations.

In the following subsections, we discuss the sub-algorithms on which the Error Detector relies. In Subsection §11.4.1, we present our algorithm to find the correlation in which a binding is involved; we then report on a number of ancillary sub-algorithms, namely: an algorithm to find the artefacts that are involved in a given correlation, *cf.* Subsection §11.4.2, an algorithm to find the sub-correlation that corresponds to a given artefact, *cf.* Subsection §11.4.3; and an algorithm to find the subset of sub-rules according to which a correlation is invalid, *cf.* Subsection §11.4.4; the previous algorithms are used to implement the algorithm to verify a correlation, which we present in Subsection §11.4.5.

11.4.1 Finding correlations

The Error Detector provides a method called `findCorrelation` whose algorithm is presented in Program §11.3. It gets a binding as input and calculates the correlation in which it is involved. The main loop navigates from the binding that is passed as a parameter to all of the bindings that are reachable from it, either directly or transitively, and to all of the bindings from which it can be reached, either directly or transitively. In other words, it implements a breadth-first search to calculate the expansion of a node in a graph [48].

For instance, if algorithm `findCorrelation` is invoked on bindings `n1`, `n4`, `n6`, or `n8` in Figure §11.8, it then would return the correlation in Figure §11.10.

```
1: to findCorrelation(in b: Binding): Correlation do
2:   result = new Correlation(nodes =  $\emptyset$ , edges =  $\emptyset$ )
3:   g = WorkGraph.getInstance()
4:   q =  $\emptyset$ 
5:   add b to q
6:   whilst q <>  $\emptyset$  do
7:     d = take a binding from q
8:     add d to result.nodes
9:     cs = (children of d in g) \ result.nodes
10:    add cs to q
11:    for each c in cs do
12:      h = new Edge(parent = d, child = c)
13:      add h to result.edges
14:    end for
15:    ps = (parents of d in g) \ result.nodes
16:    add ps to q
17:    for each p in ps do
18:      h = new Edge(parent = p, child = d)
19:      add h to result.edges
20:    end for
21:  end whilst
22:  return result
23: end
```

Program 11.3: *Algorithm to find correlations.*

11.4.2 Finding the artefacts involved in a correlation

A correlation may involve several artefacts. This situation is very common since typical integration solutions involve several processes, some of which may be shared. This implies that an event that is reported from a port may result in a binding that actually involves several artefacts.

The Error Detector provides a method called `findArtefactsInvolved` whose algorithm is presented in Program §11.4. It gets a correlation as input and returns a set of artefacts. The main loop of this method iterates over the set of bindings in the correlation that is passed as a parameter. In each iteration, it firsts finds the processes that own the ports from which the cor-

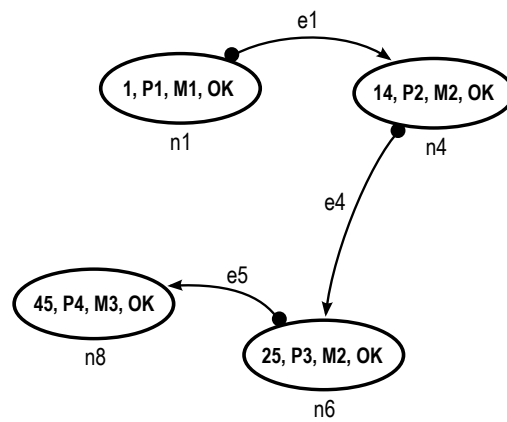


Figure 11.10: *Sample correlation.*

```

1: to findArtefactsInvolved(in c: Correlation): Set(Artefact) do
2:   result = ∅
3:   for each binding b in c.nodes do
4:     p = find the process to which a port
5:         called b.portName belongs in
6:         the Meta-information database
7:     s = find the integration solutions to which
8:         process p belongs in
9:         the Meta-information database
10:    add s ∪ {p} to result
11:  end for
12:  return result
13: end

```

Program 11.4: *Algorithm to find the artefacts involved in a correlation.*

responding events were reported, and the integration solutions in which they are involved. The loop simply adds all of these artefacts to the result, and then returns the whole collection.

For instance, if we invoke method `findArtefactsInvolved` on the correlation in Figure §11.10, the following artefacts involved would be returned: `Solution1`, `Solution2`, `Prc1`, and `Prc2`, cf. Figure §11.11.

```

1: to findSubCorrelation(in c: Correlation,
2:     in a: Artefact): Correlation do
3:   p = find all ports of artefact a in the Meta-information database
4:   b = find all bindings b in c.nodes such that binding b.portName
5:     belongs to p.portName in the Meta-information database
6:   e = find all edges x such that {x.source, x.target}  $\subseteq$  b
7:   return new Correlation(nodes = b, edges = e)
8: end

```

Program 11.5: Algorithm to find sub-correlations.

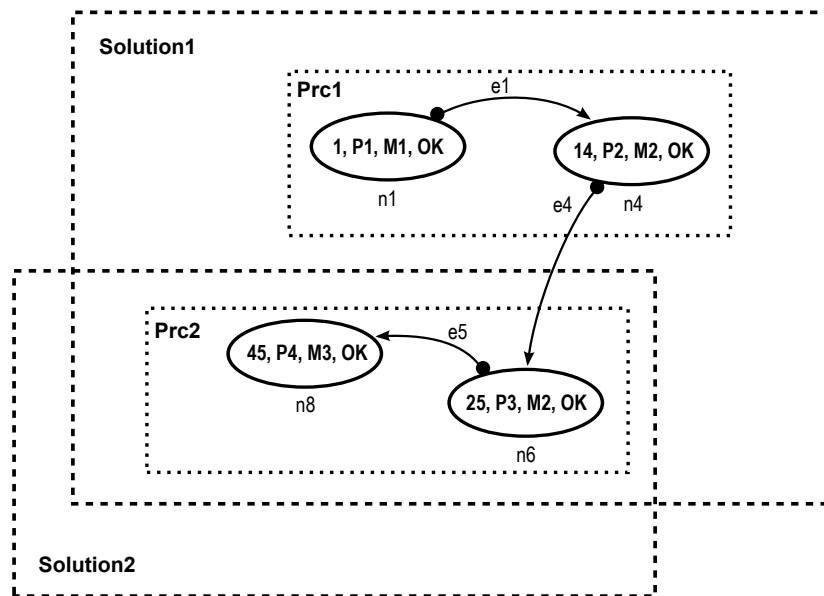


Figure 11.11: Artefacts involved in a correlation.

11.4.3 Finding sub-correlations

By sub-correlation, we refer to a subset of a correlation in which the ports involved belong to a given artefact. Note that there are not any structural differences between correlations and sub-correlations: they are both represented as graphs. In the sequel, we write sub-correlation wherever we wish to em-

```
1: to findFailingRules(in c: Correlation,
2:                   in a: Artefact): Set(Name) do
3:   result =  $\emptyset$ 
4:   for each rule r in a.rules do
5:     for each atom t in r.atoms do
6:       n = count bindings b in c.nodes
7:         such that b.portName == t.port.name
8:       if n <= t.min or n >= t.max then
9:         add r.name to result
10:      end if
11:    end for
12:  end for
13:  return result
14: end
```

Program 11.6: *Algorithm to find failing rules.*

phasise that there is a single artefact from which all of the events represented in a correlation were reported.

The Error Detector provides a method called `findSubCorrelation` whose algorithm is presented in Program §11.5. It gets a correlation and an artefact as input and returns a sub-correlation. The algorithm first finds which ports belong to the artefact, and then finds all of the bindings in the correlation whose ports are in the previous set; to create the resulting sub-correlation we just need to find the whole collection of edges that connect the previous bindings.

For instance, Figure §11.11 shows all of the sub-correlations we can find in the correlation in Figure §11.10. The dotted boxes indicate the boundary of each sub-correlation.

11.4.4 Finding failing rules

The Error Detector provides a method called `findFailingRules` that takes a sub-correlation and an artefact as input and returns a set of names that denote the rules associated with the artefact that need to be satisfied to

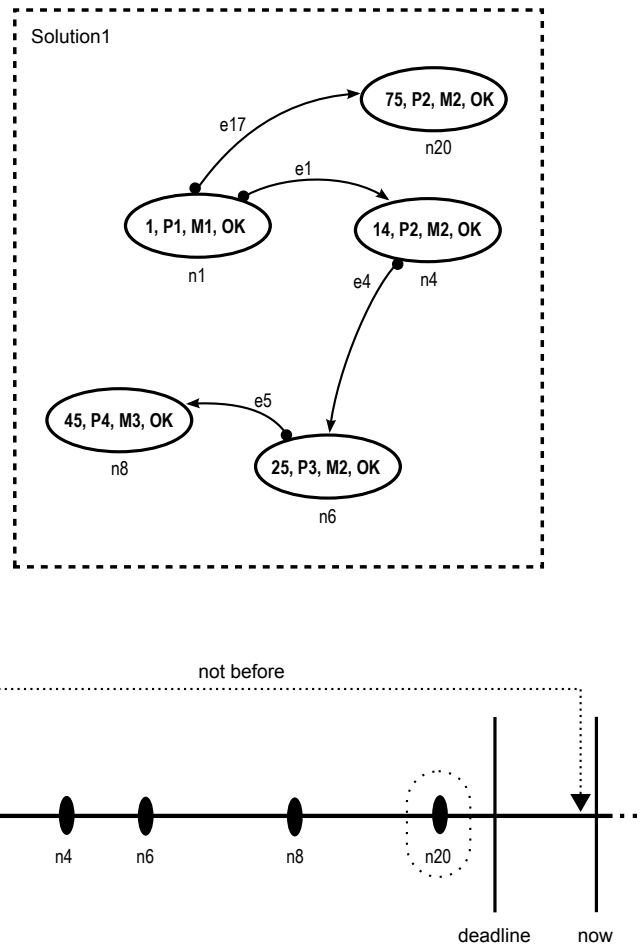


Figure 11.12: Correlation that does not satisfy a rule due to excess of bindings.

declare the correlation valid.

The algorithm to the `findFailingRules` method is presented in Program §11.6. It iterates through the collection of rules that are associated with a given artefact, and then through their atoms. The algorithm basically counts the number of bindings in the given sub-correlation that corresponds to events that were reported from the ports to which each atom refers; if this count does not lie within the limits that the atom specifies, then the corresponding rule does not validate the correlation and can thus be added to the result of the algorithm.

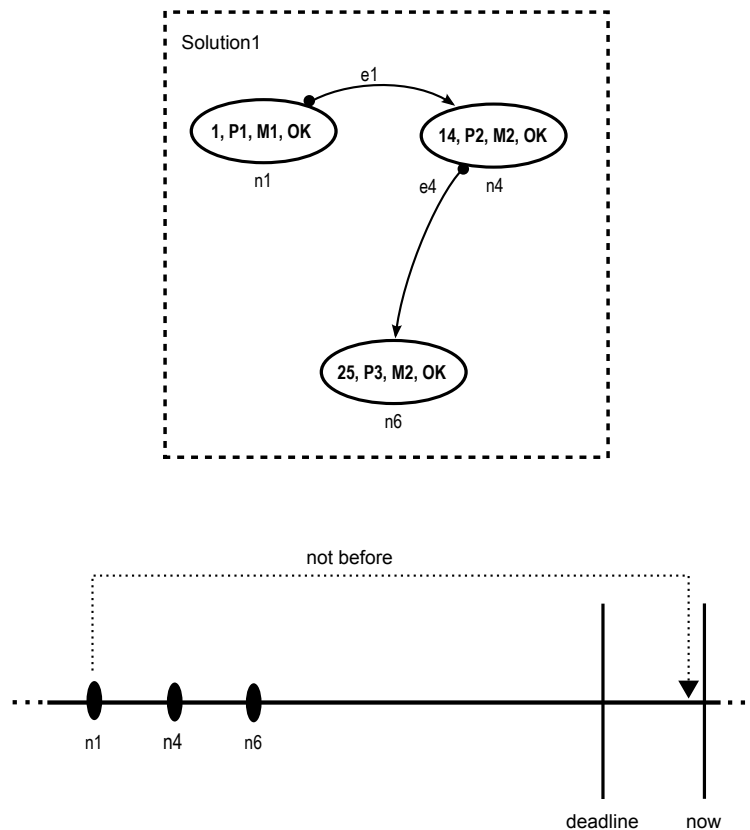


Figure 11.13: Sub-correlation that causes a rule to fail due to lack of bindings.

Figures §11.12 and §11.13 illustrate the two situations in which a correlation is considered invalid due to a failing rule. The left side of the figures represents the sub-correlation being analysed, whereas the right side represents the times at which events were reported; now represents the instant at which the analysis is performed, and deadline the latest time at which a correlation is expected to be produced (see more on this below); not before represents the time not before which the initial binding in a correlation can be analysed. Consider, for example, rule R1 for artefact `Solution1`, which was introduced in Figure §11.6; it states that zero or one correlated bindings are expected at port P2 for each binding at port P1. Note that in the sub-correlation in Figure §11.12 there are two correlated bindings at port P2, namely n4 and n20, which causes rule R1 to fail due to excess of bindings. Contrarily, in Figure §11.13 the sub-correlation contains less bindings than expected.

```

1: to verifyCorrelation(in c: Correlation) do
2:   artefacts = findArtefactsInvolved(c)
3:   for each artefact a in artefacts do
4:     s = findSubCorrelation(c,a)
5:     status = for all binding b in s, b.status == Status::OK
6:     earliestInstant = minimum of s.nodes.instant
7:     latestInstant = maximum of s.nodes.instant
8:     deadline = earliestInstant + a.timeOut
9:     isValid = (latestInstant <= deadline) and
10:              (status == true) and
11:              findFailingRules(s,a) <> a.rules
12:     if not isValid then
13:       n = new Notification(artefactName = a.name,
14:                           correlation = c)
15:       send n to the next fault-tolerance stage
16:     end if
17:   end for
18: end

```

Program 11.7: *Algorithm to verify correlations.*

11.4.5 Verifying correlations

Verifying (sub)correlations is one of the central tasks of the Error Detector. A sub-correlation can be either valid or invalid. It is valid if all of its bindings have status `Status::OK`, at least one rule does not fail to validate it, and the messages to which it refers where read or written within a given deadline; otherwise, it is invalid. The deadline refers to the time when the first message in a correlation was read or written plus the time out of the corresponding artefact. Recall that each artefact is associated with a time out that represents the maximum time it is expected to produce a correlation, cf. Figure §11.3.

The Error Detector provides a method called `verifyCorrelation` to perform this task. The algorithm to this method is presented in Program §11.7. It takes a correlation as input and if the current correlation is invalid, then it produces a notification. The algorithm first calculates the artefacts involved in the correlation and iterates through them to find their corresponding

Notation	Meaning
s	Maximum number of integration solutions to which a process can belong.
u	Maximum number of source bindings in an event.
b	Maximum number of bindings in a correlation.
c	Maximum number of child bindings of a given binding.
p	Maximum number of parent bindings of a given binding.
r	Maximum number of rules associated with an artefact.
t	Maximum number of atoms in a rule.
a	Maximum number of artefacts involved in a correlation.
n	Number of entries in the Work Queue.

Table 11.1: Notation used in our error-detection complexity analysis.

(sub)correlations; each sub-correlation is checked for validity according to the definition in the previous paragraph. (Sub)correlations that are found invalid are transformed into notifications that are sent to the following fault-tolerance stage so that they can be diagnosed.

11.5 Complexity analysis

In this section, we analyse our proposal and characterise its complexity. It can deal with an arbitrary number of processes and integration solutions, but we assume that there is a sensible upper bound; this does not amount to loss of generality since the number of artefacts of which a company's software ecosystem is composed must necessarily be finite.

Table §11.1 summarises the notation we use in this section. Our analysis proves that our proposal is computationally tractable since handling events runs in $O(1)$ time and detecting errors runs in $O(\log n)$ time for a given software ecosystem. This makes the proposal theoretically appealing since it is logarithmic on the size of the Work Queue database, which is expected not to be monotonically increasing or decreasing, but to grow and shrink as time progresses. Our experiments support this conjecture, cf. Section §11.6.

11.5.1 On the implementation

The Meta-information database is a simple set of data. None of our queries involve joining information from other databases. In our prototype we use a hash function to index the entries of the Meta-information database, and we have implemented maps from port names onto processes, processes onto integration solutions, and so on. The space required by this design is proportional to the number of artefacts, ports, and rules. As a conclusion, it is possible to retrieve information from the Meta-information database in $O(1)$ time.

The Work Queue database relies on a Brodal [12] priority queue, which allows to insert entries or retrieve the next to be analysed in $O(1)$ time, whereas removing an entry takes $O(\log n)$ time, where n denotes the size of the queue.

11.5.2 Handling events

We first report on the complexity of the algorithm to handle events, and prove that it is computationally tractable because it runs in constant time for a given software ecosystem.

Theorem 11.1 *Algorithm `handle` in Program §11.1 terminates in $O(s+u)$ time.*

Proof Handling an event involves finding a process using a port name and the integration solutions to which this process belongs. Finding this information can be accomplished in $O(1)$ time in our implementation (lines §3 and §5). The computation of the maximum time out can be accomplished in $O(s)$ time (line §7). Getting the Work Graph and the Work Queue instances can be accomplished in $O(1)$ time (lines §8 and §9). Lines §10 and §11 add the target binding to the Work Graph and to the Work Queue databases, respectively, which can also be accomplished in $O(1)$ time. The loop in lines §12–§15 iterates u times at most. Lines §13 and §14 can be implemented in $O(1)$ time since they just create an object and add it to a set. As a conclusion, the algorithm `handle` terminates in $O(s + u)$ time. \square

Corollary 11.1 *In a given software ecosystem, there must be an upper bound to $s + u$ because the number of integration solutions in a company's software ecosystem is finite, which in turn implies that there is an upper bound to the number of source bindings in an event. As a conclusion algorithm `handle` terminates in $O(1)$ time in a given software ecosystem.*

11.5.3 Detecting errors

We now analyse the complexity of the algorithm to detect errors. Note that this algorithm does not terminate, since it is a never ending loop. In this case, the complexity refers to the complexity of an iteration of this loop.

Theorem 11.2 *Every iteration of Algorithm `detectErrors` in Program §11.2 terminates in $O(b(1 + c + p + a + \log n) + a r t)$ time.*

Proof In each iteration, the algorithm first fetches an entry from the Work Queue database, which is accomplished in $O(1)$ time in our implementation, cf. Section §11.2 (line §4). According to Theorems §11.3 and §11.4 below, lines §5 and §6 run in $O(b(c + p))$ and $O(b + a(b + r t))$ time, respectively. The loop in lines §7–§9 iterates b times at most; in each iteration, line §8 removes a binding from the Work Queue database, which is accomplished in $O(\log n)$ time. As a conclusion, each iteration of Algorithm `detectErrors` terminates in $O(1 + b(c + p) + b + a(b + r t) + b \log n) = O(b(1 + c + p + a + \log n) + a r t)$ time. \square

Corollary 11.2 *In a given software ecosystem, b , c , p , a , r , and t are constants because there is an upper limit to the number of artefacts a company runs. As a conclusion, each iteration of Algorithm `detectErrors` terminates in $O(\log n)$ time in a given software ecosystem.*

Next, we analyse the complexity of Algorithm `findCorrelation`.

Theorem 11.3 *Algorithm `findCorrelation` in Program §11.3 terminates in $O(b(c + p))$ time.*

Proof The loop in lines §6–§21 iterates b times at most. In each iteration, the algorithm executes two inner loops. The first one, iterates c times at most and the operations inside terminate in $O(1)$ time since they just involve creating objects and adding them to a set. Similarly, the second one iterates p times at most and the operations inside also terminate in $O(1)$ time. As a conclusion, Algorithm `findCorrelation` terminates in $O(b(c + p))$ time. \square

Now, we analyse the complexity of Algorithm `verifyCorrelation`.

Theorem 11.4 *Algorithm `verifyCorrelation` in Program §11.7 terminates in $O(b + a(b + r t))$ time.*

Proof The algorithm first calculates the number of artefacts involved in a given correlation, which terminates in $O(b)$ time according to Theorem §11.5 below (line §2). It then executes a loop that iterates a maximum of a times (lines §3–§17). In each iteration, it first needs to find a number of sub-correlations, which terminates in $O(b)$ time according to Theorem §11.6 (line §4); then, it calculates the status, the earliest instant, and the latest instant, which requires iterating a maximum of b times (lines §5–§7); calculating the deadline can be accomplished in $O(1)$ time (line §8), but determining if a sub-correlation is valid involves executing algorithm `findFailingRules`, which runs in $O(rt)$ time according to Theorem §11.7 below. Lines §12–§16 run in $O(1)$ time since they just require creating an object and sending it to another stage. As a conclusion, Algorithm `verifyCorrelation` terminates in $O(b + a(b + rt))$ time. \square

In the following theorems, we analyse the complexity of the three sub-algorithms on which `verifyCorrelation` relies.

Theorem 11.5 *Algorithm `findArtefactsInvolved` in Program §11.4 terminates in $O(b)$ time.*

Proof The loop at lines §3–§11 iterates b times at most. Lines §5–§10 can be implemented in $O(1)$ time, cf. Section §11.2. As a conclusion, finding the artefacts involved in a given correlation terminates in $O(b)$ time. \square

Theorem 11.6 *Algorithm `findSubCorrelation` in Program §11.5 terminates in $O(b)$ time.*

Proof The algorithm first finds the ports that belong to a given artefact, which can be accomplished in $O(1)$ time (line §3). It then finds the bindings in a correlation whose port belongs to the previous set, which requires iterating through the bindings in the correlation (line §3); this requires $O(b)$ time in the worst case. Finally, the algorithm finds the edges that connect the previous bindings, which also requires $O(b)$ time (line §6), and creates an object in $O(1)$ time (line §7). As a conclusion, Algorithm `findSubCorrelation` terminates in $O(1 + 2b) = O(b)$ time. \square

Theorem 11.7 *Algorithm `findFailingRules` in Program §11.6 terminates in $O(rt)$ time.*

Proof The loop at lines §4–§12 iterates a maximum of r times, and the inner loop at lines §5–§11 iterates t times at most. As a conclusion, Algorithm `findFailingRules` runs in $O(rt)$ time. \square

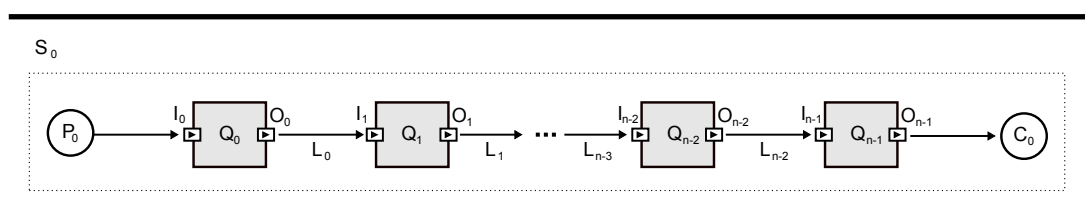


Figure 11.14: *The Pipeline pattern.*

11.6 Fault tolerance experiments

We have conducted a series of experiments to evaluate our proposal in the laboratory. We implemented them on top of a discrete event simulation layer that allowed us to run the experiments in simulated time. We ran the experiments on a machine equipped with a four-core Intel Xeon processor running at 3.00 GHz, and had 16 GB of RAM, Windows Server 2008 64-bit, and the Java Enterprise Edition 1.6 installed.

In the following sections, we first provide additional details on the patterns that we have used in our experiments and then on the experimentation parameters; later, we draw our conclusions about the experimental results we gathered.

11.6.1 Experimentation patterns

We set up six well-known patterns that lie at the core of most real-world integration solutions, namely: pipeline, dispatcher, merger, request-reply, splitter, and aggregator [54]. In the sequel, we use term producer to refer to the process or application that produces the messages that are fed into a pattern; similarly, we use term consumer to refer to the process or application that consumes the messages that the pattern produces. Furthermore, we use variable n to refer to the total number of processes involved in each pattern.

In the pipeline pattern, messages flow from a producer to a consumer in sequence: the messages a process produces are consumed by the next process in the pipeline, *cf.* Figure §11.14. There is a single producer and a single consumer, *i.e.*, changes to n have an impact on the number of intermediate processes only. In other words, $2n$ events are reported to the monitor per message the producer feeds into the pattern. The number of events depends on

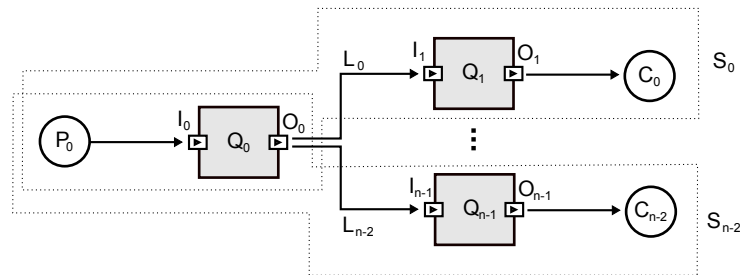


Figure 11.15: *The Dispatcher pattern.*

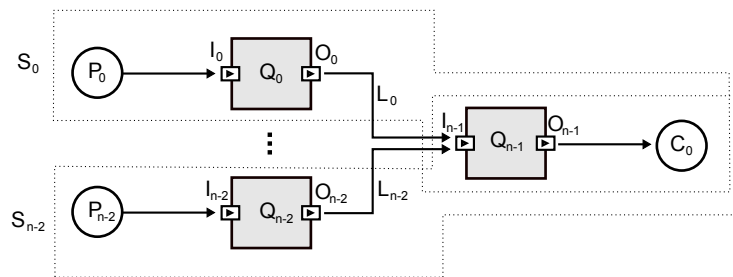


Figure 11.16: *The Merger pattern.*

the number of ports a message goes through in the pattern, each port notifies an event.

In the dispatcher pattern, there is a process that routes the messages it produces to only a specific consumer, *cf.* Figure §11.15. Note that changes to n have an effect on the number of consumers, not on the number of producers, which is one. That is, 4 events are reported to the monitor per message the producer feeds into the pattern.

The goal of the merger pattern is to gather messages from several producers and route them to a unique consumer, *cf.* Figure §11.16. Changes to n have an impact on the number of producers, not on the number of consumers, which is one. Note that 4 events are reported to the monitor per message a producer feeds into the pattern.

In the request-reply pattern, there are a number of processes that require a service from another process, *cf.* Figure §11.17. Changes to n have an effect on

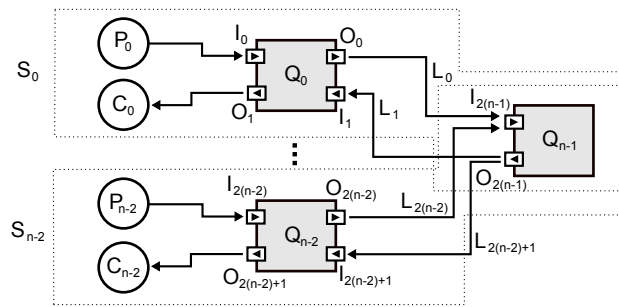


Figure 11.17: The Request-Reply pattern.

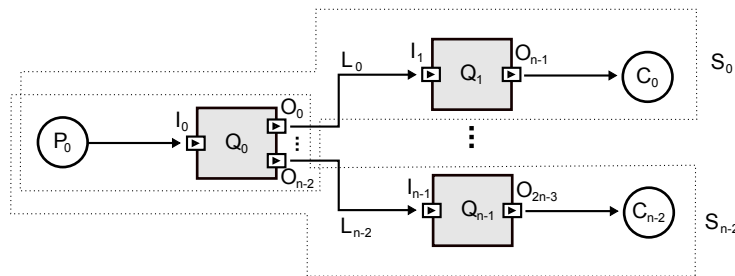


Figure 11.18: The Splitter pattern.

the number of client processes that send requests to the single server process. Every message fed into the pattern results in 6 events that are reported to the monitor.

The splitter pattern has a process that splits the messages it receives from a producer into two or more messages, each of which carries a piece of the original message to a different consumer, cf. Figure §11.18. Note that changes to n have an effect on the number of consumers, not on the number of producers, which is one. That is, $2(n - 1) + n$ events are reported to the monitor per message the producer feeds into the pattern.

In the aggregator pattern, there is a process that aggregates messages from different producers into a single message, which is made available to a unique consumer, cf. Figure §11.19. Changes to n have an impact on the number of producers, not on the number of consumers, which is one. Note that $2(n - 1) + n$ events are reported to the monitor per message delivered by the pattern.

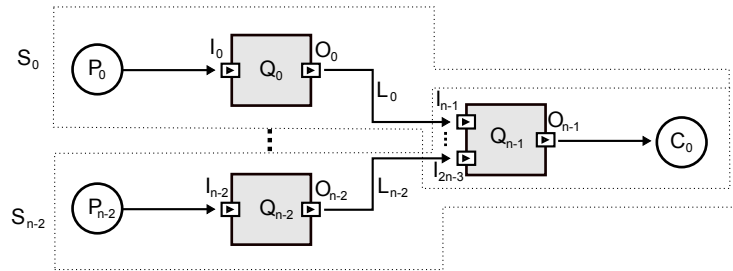


Figure 11.19: The Aggregator pattern.

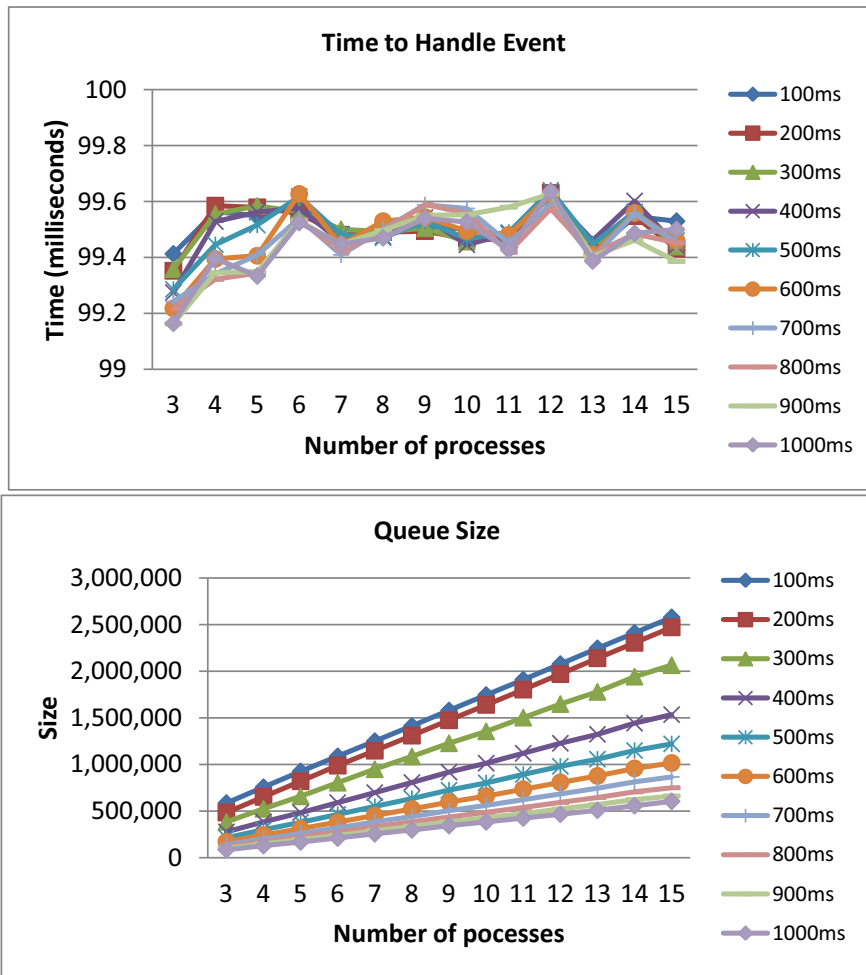


Figure 11.20: Experimental results for the Pipeline pattern.

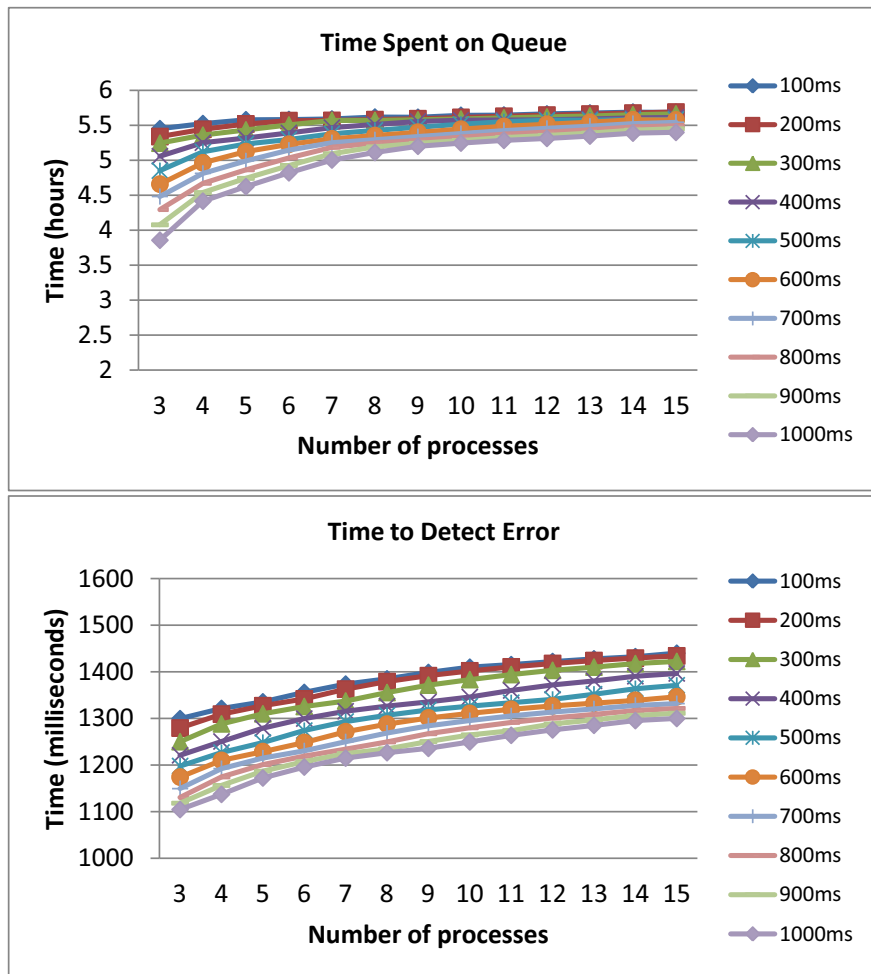


Figure 11.20: Experimental results for the Pipeline pattern (Cont'd).

11.6.2 Experimentation parameters and variables

Each experiment consisted of running an instance of a pattern with a fixed number of processes (n) and a fixed mean message production rate (t); we varied n in the range 3..15 processes, and t in the range 100..1000 milliseconds, with increments of 100 milliseconds. In total, we ran 130 experiments for each pattern to draw our conclusions.

We sampled the production rate from a negative exponential with parameter t . Similarly, we sampled both the time to transmit messages and the time

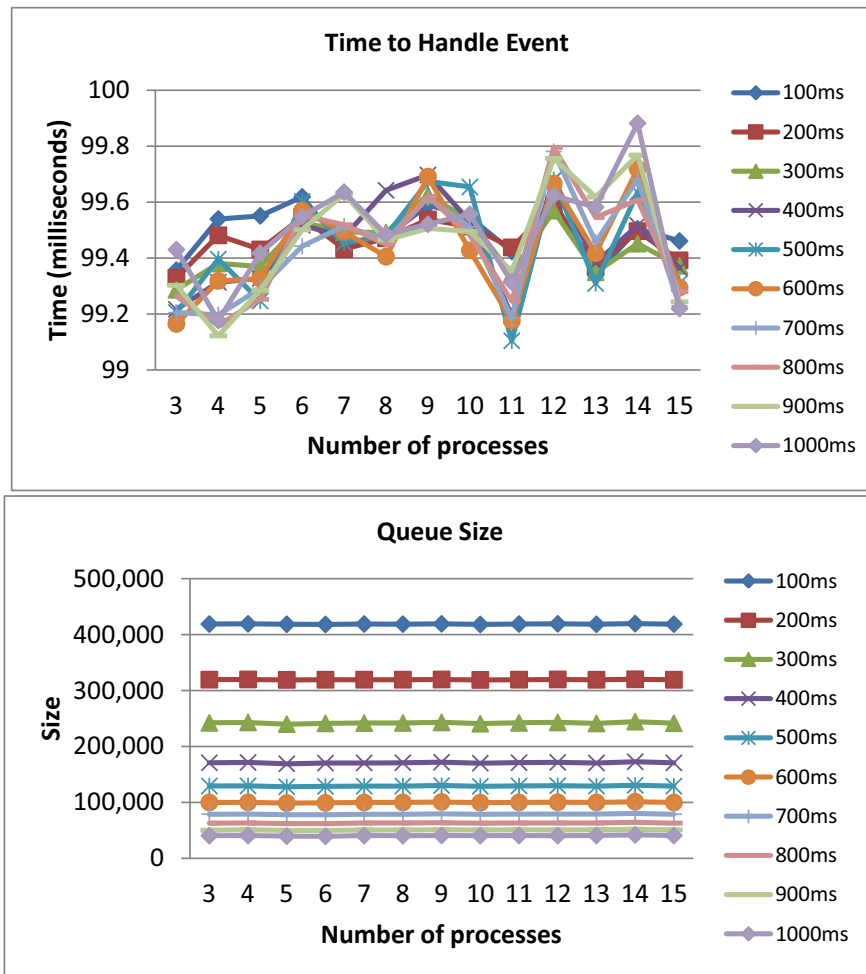


Figure 11.21: Experimental results for the Dispatcher pattern.

each process took to produce a correlation from a negative exponential with parameter 250 milliseconds; the time out of every artefact was set to 5 minutes. We carried out additional experiments with other values for these parameters and found that they did not have an impact on the conclusions. Each experiment was run for a duration of 24 hours.

In each experiment, we measured: the time to handle an event (THE), the size of the Work Queue (QS), the time each binding spent in the Work Queue after it was scheduled to be analysed (TSQ), and the time the detectError algorithm took to perform each iteration of its main loop (TDE). In the sequel,

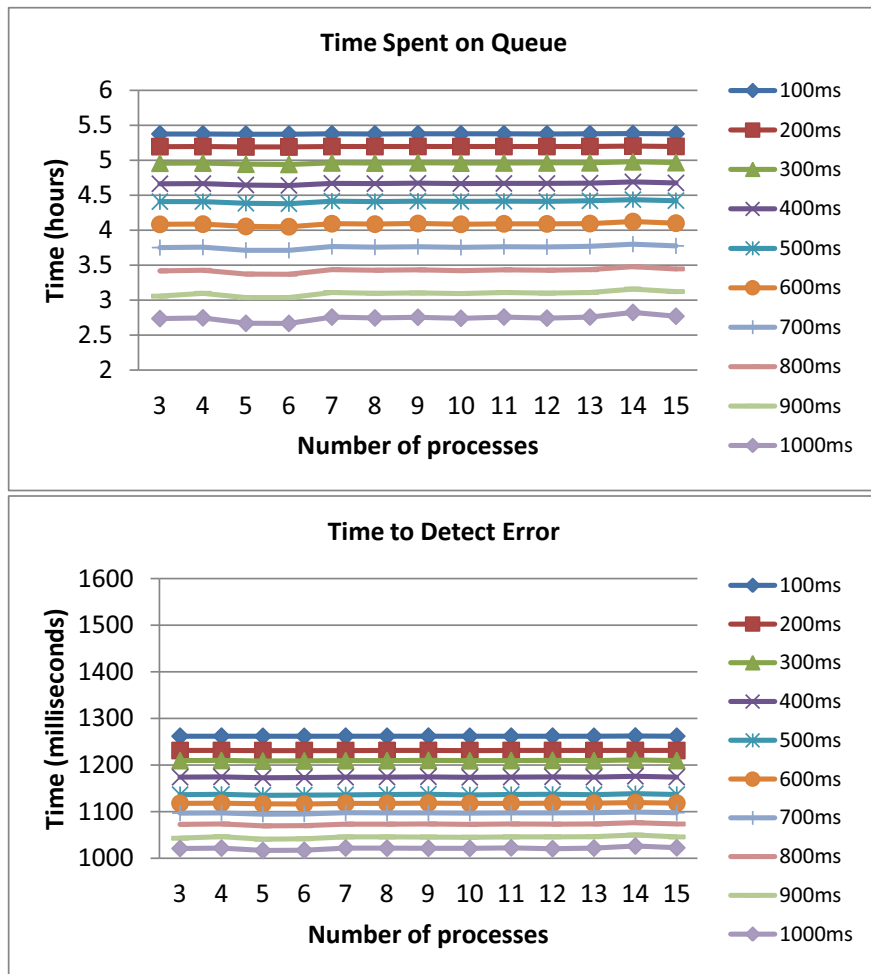


Figure 11.21: Experimental results for the Dispatcher pattern (Cont'd).

we report on the averaged values of these variables after discarding a few outliers using a method based on the well-known Chevischev inequality.

11.6.3 Experimentation results

Figures §11.20, §11.21, §11.22, §11.23, §11.24, and §11.25 present the results we have gathered regarding variables THE, QS, TSQ, and TDE for each of the patterns we presented before.

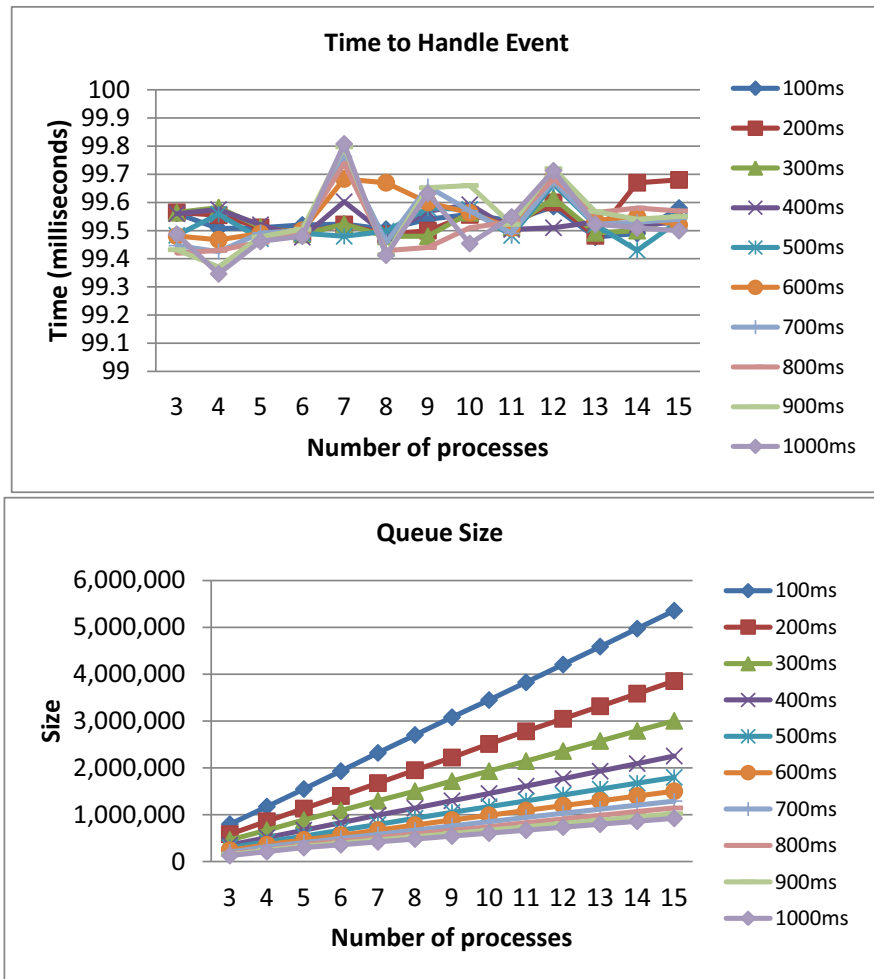


Figure 11.22: Experimental results for the Merger pattern.

The time to handle events (THE) remains low in all of the experiments, and the changes to n or t do not seem to have an impact on it. According to Theorem §11.1, the time to handle an event depends on the maximum number of integration solutions to which a process can belong and on the maximum number of source bindings in an event, which are fixed constants for a given ecosystem. In Corollary §11.1, we argued that there is an upper bound to these figures, and we concluded that the time to handle events might be considered $O(1)$ in practice. The experiments corroborate this idea, since THE seems to be totally independent from n or t in practice; note that it varies largely, but in a short interval to which there seems to be a clear upper bound in every case.

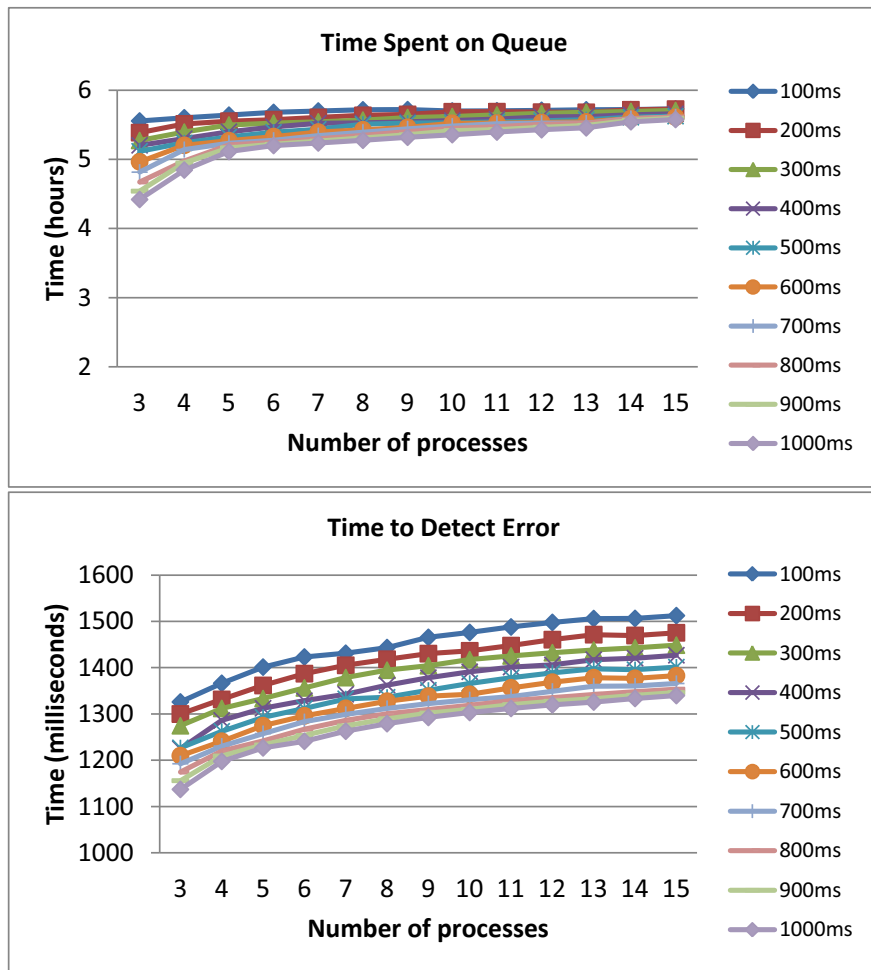


Figure 11.22: Experimental results for the Merger pattern (Cont'd).

The size of the `Work Queue (QS)` is important insofar as it has an impact on the memory footprint (the larger the queue the more memory is consumed) and the time required to complete an iteration of `Algorithm detectErrors` (recall that this algorithm removes all of the bindings that are correlated with the binding being analysed from the `Work Queue`, which requires $O(|QS|)$ time). It depends on the number of events reported in each experiment. In most cases, it behaves linearly with respect to the number of processes with a slope that depends on the mean message production rate (it decreases as this parameter increases). The only exception is the dispatcher pattern, in which `QS` seems to be a constant that depends on the mean message production rate

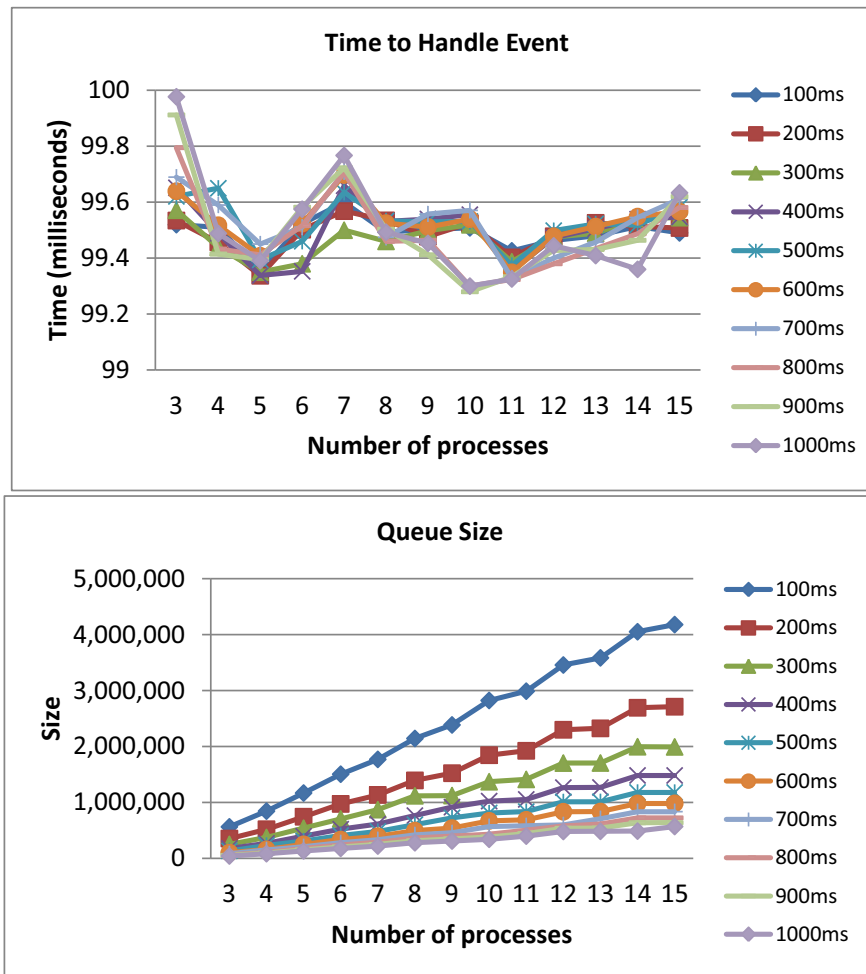


Figure 11.23: Experimental results for the Request-Reply pattern.

only. The reason for this behaviour is that changes to n do not have an impact on the number of events that are reported. Note that every new process in the pipeline pattern contributes with 2 additional events, new processes in the merger pattern contribute with 4 additional events, and new processes in the request-reply pattern contribute with 6 additional events. Contrarily, adding a new process to the dispatcher pattern does not contribute with new events. The reason is that adding a new consumer implies that there is a new process that competes for the messages the producer feeds into the pattern; in other words, the events are reported from different sources, but the total number of events remains constant.

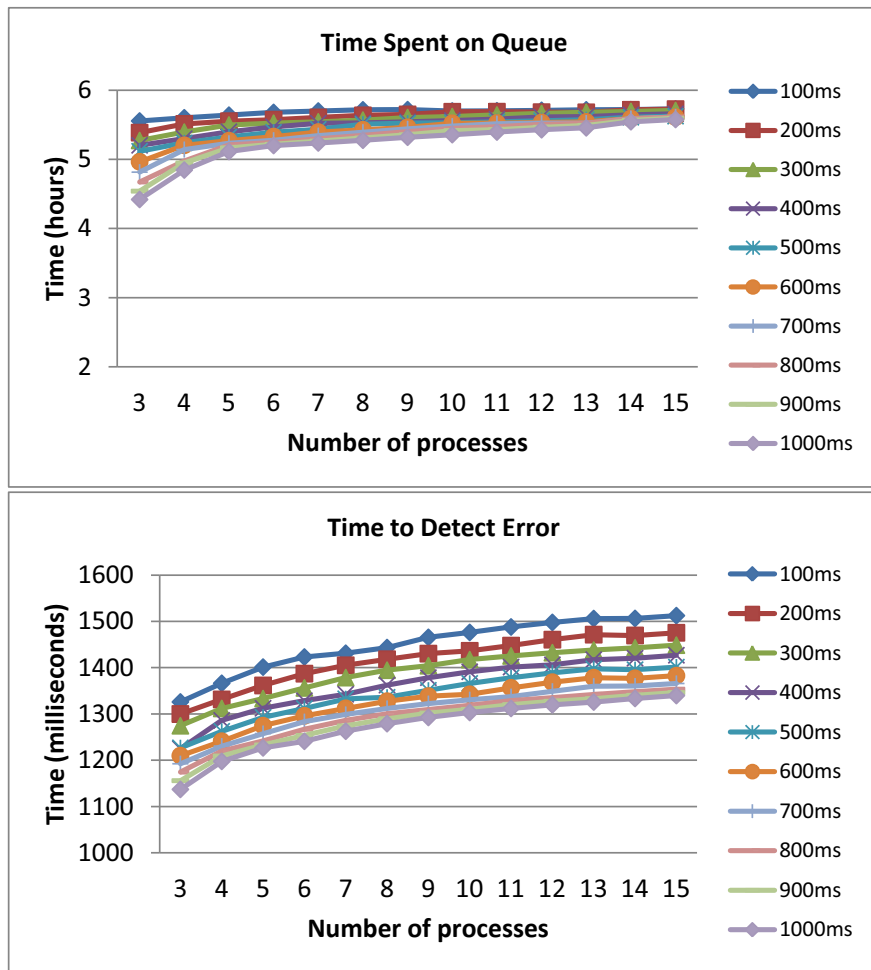


Figure 11.23: Experimental results for the Request-Reply pattern (Cont'd).

Variable TSQ is the most relevant to draw our conclusions. Recall that this variable measures the time a binding spends in the Work Queue after it is scheduled to be analysed by the Error Detector. The length of the time spent in the queue does not affect the correct functionality of the monitor. However, generally speaking, the less time, the better since this implies that errors shall be detected, diagnosed, and recovered as soon as possible. Our experiments prove that TSQ seems to behave logarithmically in the number of processes in all cases, except for the dispatcher pattern, in which it behaves constantly. This implies that a change to the number of processes does not usually have a significant impact on the time bindings spend in the Work Queue. Neither

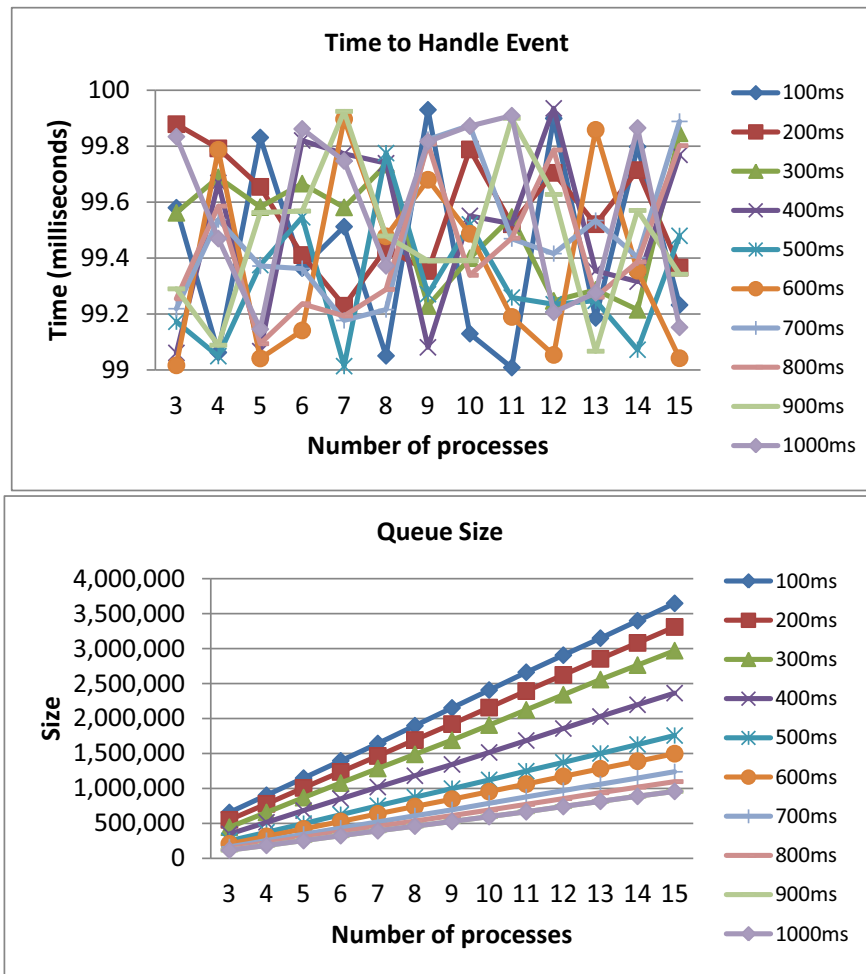


Figure 11.24: Experimental results for the Splitter pattern.

does the mean message production rate seem to have a negative impact on this variable, which is, obviously, a good piece of news.

Regarding the time to detect errors (TDE), we proved that it behaves logarithmically in the size of the Work Queue, cf. Theorem §11.2. This theoretical result is promising as long as the size of the Work Queue does not increase monotonically, as we conjectured in Section §11.5. Our results support this conjecture since variable TDE ranges in the order of seconds in all of our experiments, even in the case of heavily-loaded scenarios with 15 processes and a message production rate of 100 milliseconds.

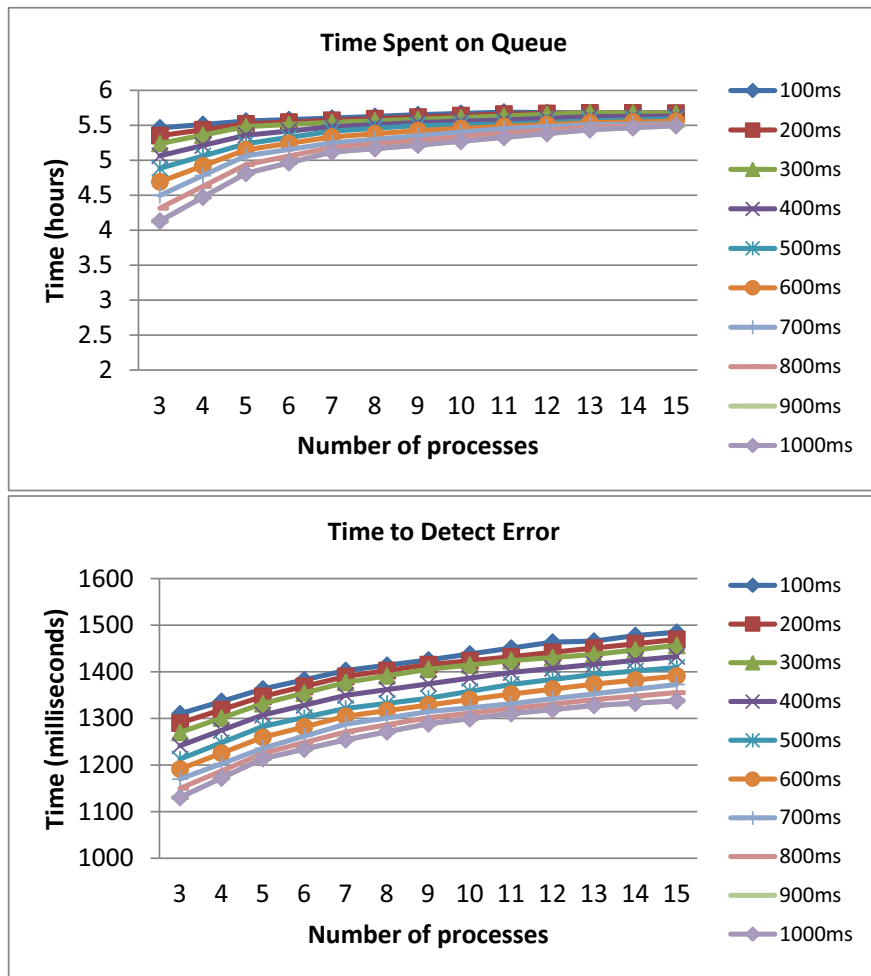


Figure 11.24: Experimental results for the Splitter pattern (Cont'd).

11.7 Summary

In this chapter, we have reported on a monitor that receives information about the messages that are read or written at each port and uses them to build a graph that keeps record of the messages exchanged within an integration solution. We have designed and implemented the algorithms to analyse this graph and detect potential errors introduced when messages are read and written by ports. Furthermore, we have analysed our proposal from a theoretical point of view and the results have been corroborated in the laboratory.

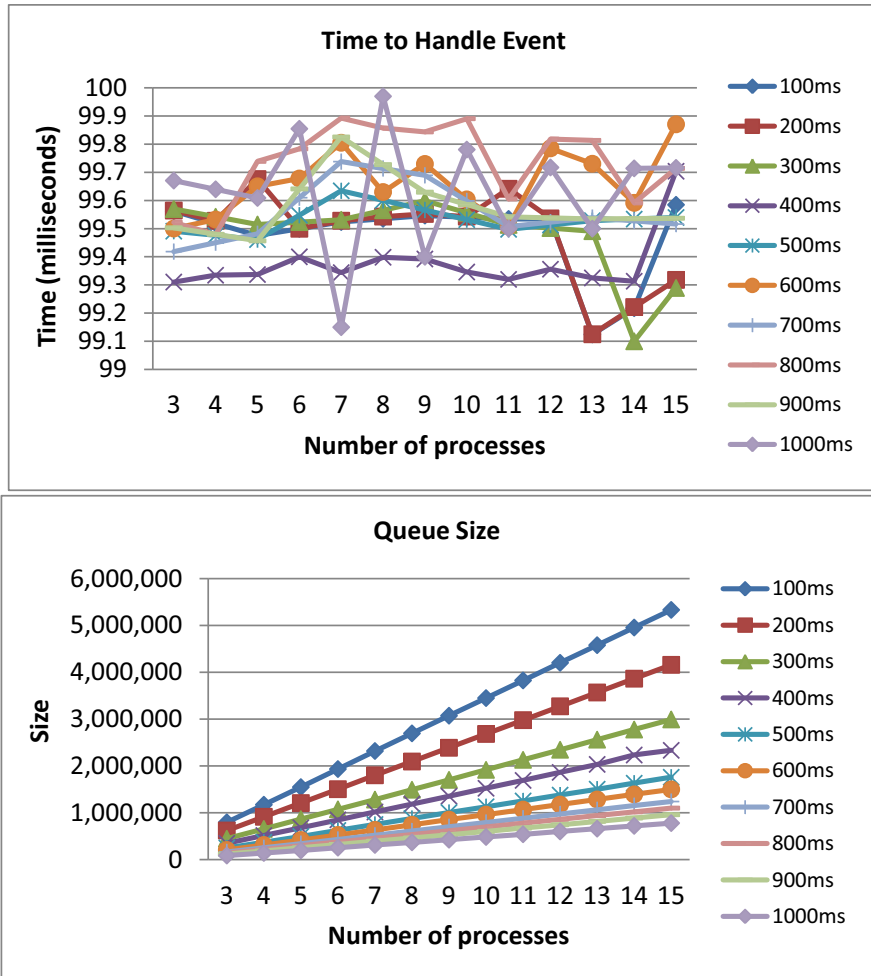


Figure 11.25: Experimental results for the Aggregator pattern.

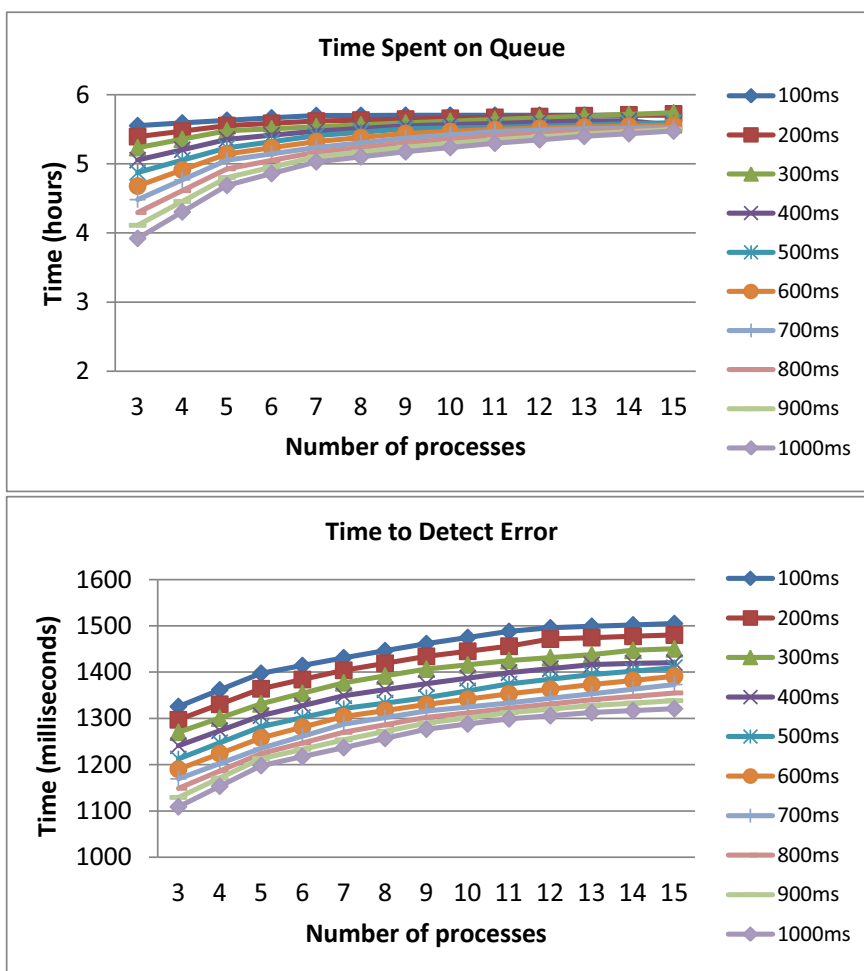


Figure 11.25: Experimental results for the Aggregator pattern (Cont'd).

Chapter 12

Case Studies

*Knowing is not enough; we must apply.
Willing is not enough; we must do.*

Johann W. von Goethe, German Novelist (1749–1832)

Applying our proposal to real-world case studies was very important to verify it and prove that it is viable. Section §12.1 introduces the case studies on which we have worked to prove that our results are practical and can be used to solve real-world problems. We provide additional details in Sections §12.2 – §12.6. Section §12.7 reports on additional results that we gathered in an experiment in which we run one of the case studies using JBI binding components. Finally, Section §12.8 summarises the chapter.

12.1 Introduction

We have worked on five case studies to which we have applied our proposal in the laboratory to verify its viability. Every case study was designed using our Domain-Specific Language and implemented using our Software Development Kit. The first case study is Café, which is an illustrative integration problem used by the Enterprise Application Integration community to demonstrate and compare integration tools; the following two case studies are based on real-world integration problems found at Unijuí University (Ijuí, Brazil) and at Huelva's County Council (Huelva, Spain); finally, we have implemented two well-known case studies in the literature to illustrate the integration of the applications involved in the scenarios of searching and booking flights and hotels for a travel.

We have conducted a series of experiments to evaluate the previous integration solutions on our Runtime System. We used mock adapters, *i.e.*, adapters implemented in memory that simulate the functionality of a real-world adapter and not on external software. The mock adapters allowed us to save the processing time required by the real-world adapters based on JBI, which have proven to have a non negligible impact on the consumption of CPU time, *cf.* Section §12.7. Furthermore, by using mock adapters the execution of the integration solutions depend only on our Runtime System, and not on other external software. In each experiment we measured the following variables:

Consumption of CPU Time per Thread: This variable measures the average CPU times that the integration solution has consumed to process all of the messages of an experiment. Note that we measured CPU time per thread, *i.e.*, the actual time the available threads took to process the workload, including user and operating system time. To measure this variable, we run every integration solution with a fixed message production rate, a varying the number of threads (t), and a varying number of messages (m). We introduced a 60-second delay between every two experiments. The message production rate considered was one message every 5 milliseconds, we varied t in the range 1, 2, 4, 6, 8 threads, and m in the range 20,000, 40,000, 60,000, ... , 200,000 messages. In total, we ran a total of 125 experiments for each integration solution to draw our conclusions on this variable.

Pending Messages: This variable measures the number of messages that had not been processed yet right after the message production finishes. The

experiments conducted to measure this variable consisted of running an integration solution with a fixed number of messages per experiment, a varying number of threads (t), and a varying message production rate (r) to simulate heavily-loaded scenarios. We introduced a 60-second delay between every two experiments. The total number of messages in each experiment was 100,000, we varied t in the range 1, 2, 4, 6, 8 threads, and r in the range 200, 400, 600, ... 3,000 messages per second. In total, we ran 375 experiments for each integration solution to draw our conclusions on this variable.

We ran these experiments on a machine that was equipped with an Intel Core i7 with four physical CPU threads that run at 2.93 GHz, and had 8 GB of RAM, Windows 7 Professional Service Pack 1, and Java Enterprise Edition 1.6 64-bit installed. Each experiment was repeated 5 times and the results were averaged in order to diminish the effects of unpredictable events in the operating system. In every experiment the body of the messages hold an actual document in XML format. Note that the size of a message being processed by an integration solution varies, since it is modified and transformed throughout the workflow. Thus, we have computed the average size of the messages that belong to a same correlation processed in an integration solution. The result is an average message size of 1,376.40 bytes for the Café solution, 1,356.14 bytes for the Unijuí University solution, 1,317.75 for the Huelva's County Council solution, 1,498.11 bytes for searching flights and hotels solution, and 1,435.66 for the booking flights and hotels solution. (Note that this deviates largely from other experiments in the literature in which the authors used unrealistically small message sizes.)

In the following sections, we first describe the integration problem tackled in each case study; we then provide a solution model; next, we provide the rules used to detect possible errors when running these integration solutions; finally, we show the experimental results that we gathered and draw our conclusions.

12.2 Café

In the literature, the Coffee Shop case study (Café for short) [53] has become the *de facto* standard to compare proposals in the integration field from a practical point of view.

12.2.1 The software ecosystem

The workflow in Café, describes how customer orders are processed in a coffee shop. Roughly speaking, it starts by a customer placing an order to the cashier, who then registers the order in the system and adds it to an order queue. An order may include entries for hot and cold drinks, which are prepared by different baristas. When all of the drinks that correspond to the same order have been prepared, they are ready to be delivered by the waiter. Every order has a tray associated to it, which is used to deliver the order to the customer. Note that, the cashier is decoupled from the baristas, since the orders taken from customers are placed in the queue from which baristas retrieve them. It allows the cashier to keep taking orders from customers even when the baristas are backed up. Baristas do not have a complete view of the whole set of drinks in an order, they receive individual drink requests and when a drink is prepared the barista places it on the corresponding tray.

The goal of the integration solution is to take orders from the order queue, send requests to the corresponding baristas to prepare the corresponding hot and cold drinks, and notify the waiter when an order is completed.

12.2.2 Solution

Figure §12.1 presents a design for this integration case study using our Domain-Specific Language. The integration solution is composed of one orchestration process that exogenously co-ordinates the four applications involved. The communication with the `Orders` and `Waiter` applications is carried out by means of ports that read and write messages; on the other hand the communication with `Barista Cold Drinks` and `Barista Hot Drinks` is carried out by means of a programming API that these applications export.

The workflow begins at entry port P1, which periodically reads the `Orders` application log to find new customer orders. Every order results in a message with the drinks to be prepared added to slot S1. Task T1 splits every message into several other messages, one for each drink. Messages are now routed by the dispatcher task T2 either towards the `Barista Cold Drinks` or `Barista Hot Drinks` applications. Task T3 replicates messages to the `Barista Cold Drinks`, so that one copy can be used to request this application to prepare the drink, by means of ports P2 and P3, and the other waits for the correlated response that brings information about the preparation of the drink.

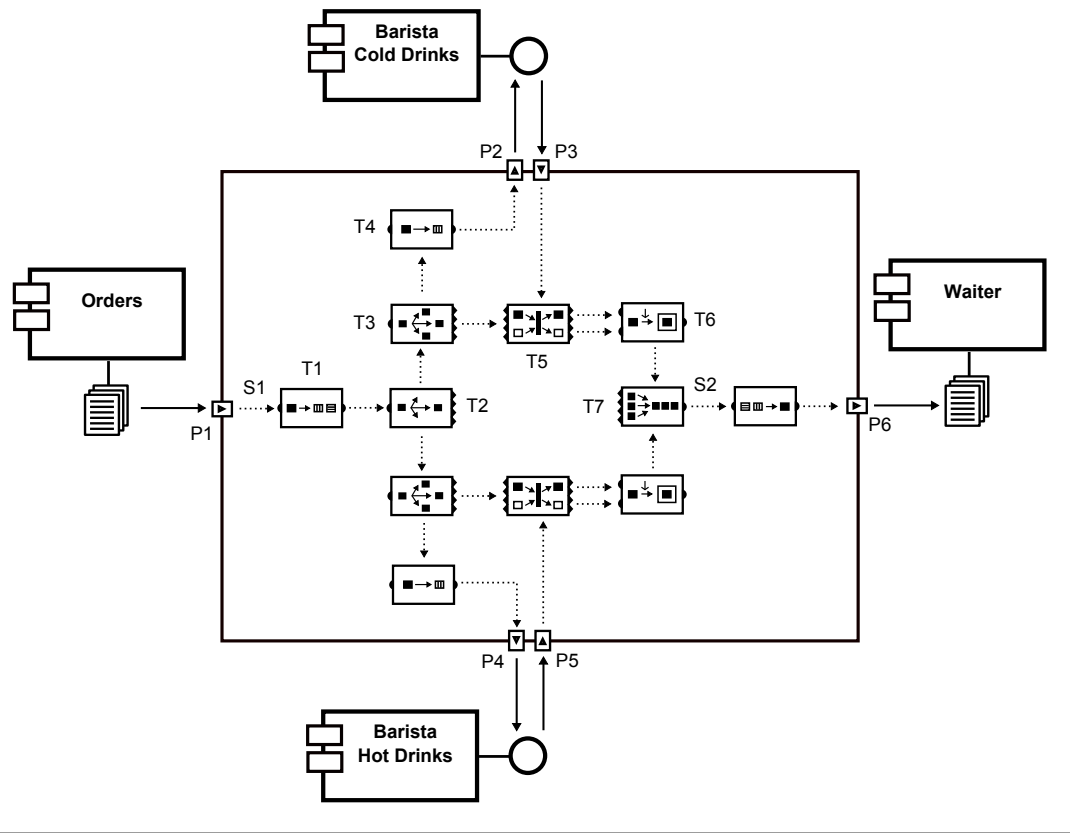


Figure 12.1: *The Café integration solution.*

Task T5 correlates the response from the barista with the waiting copy, and task T6 enriches the waiting copy with the information returned by Barista Cold Drinks. Task T4 transforms messages into the necessary request format for the application. Messages towards Barista Hot Drinks behave symmetrically. Once the drink is prepared, the messages from baristas are merged into a single slot S2 by the merger task T7. Then, the prepared drinks are taken from this slot and aggregated back into a single message for the order, so that exit port P6 writes the resulting message to the Waiter application.

12.2.3 Error detection rules

Figure §12.2 shows the rules that we have designed for the Café integration solution. Rule R1 states that for every input message read at port P1,

$$\begin{aligned} R1 &= P1[1] \longrightarrow P6[1] \\ R2 &= P2[1] \longrightarrow P3[1] \\ R3 &= P4[1] \longrightarrow P5[1] \end{aligned}$$

Figure 12.2: *Error detection rules for the Café solution.*

the integration solution must produce another correlated message at port P6. Rules R2 and R3 state that for every request message to the baristas, there must be a correlated response message. Note that we assume, that every order has at least one item; this does not amount to loss of generality since it actually does not make sense to place an order with zero drinks.

12.2.4 Experimental results

Figure §12.3 presents our experimental results. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since R^2 was 0.999. Furthermore, the graph depicted for this variable shows that the consumption of CPU time per thread reduces considerably when adding more threads until the limit of 4 threads. Then, adding more threads apparently does not reduce the consumption of CPU time per thread. The reason for this behaviour is the limit of four physical CPU threads in the processor.

The graph depicted to show the number of pending messages, indicates that the integration solution supports a message production rate r until 800 messages per second when using 4, 6, or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads with which we have experimented. If the message production rate r ranges from 1,600 to 3,000, then there is not much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if the integration solution is run with only 1 thread. If the integration solution is run with 2 threads, no messages are accumulated until $r = 600$. The behaviour is similar when using 4–8 threads, which is due to the limit of four physical CPU threads in the processor.

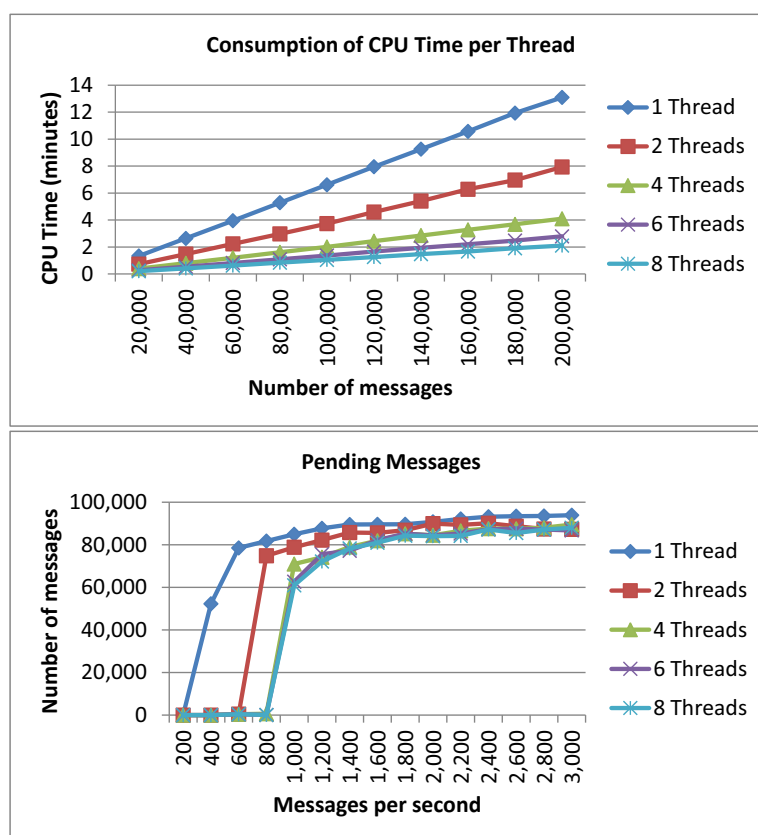


Figure 12.3: *Experimental results for the Café solution.*

12.3 Unijuí University

This case study consists of a non-trivial, real-world integration problem that builds on a project to enhance the functionality of the call centre application at Unijuí University. The goal is to automate the invoicing of personal phone calls that employees make using the University's phones.

12.3.1 The software ecosystem

The integration solution involves five applications, namely: Call Centre, Human Resources System, Payroll System, Mail Server, and SMS Notifier. Each application runs on a different platform; the Human Resources System

and the Payroll System are legacy systems developed in house, and the rest are off-the-shelf software packages purchased by the University. In addition, Mail Server provides a POP3 and a SMTP interface; the other applications were designed without integration concerns in mind, which requires to interact with them by means of their data layer. The Call Centre records every call every employee makes from a University's phone; it can identify who the employee is because they have a personal access code that they have to enter before dialling the number they wish to call. This code is used to correlate phone calls with the information in the Human Resources System and the Payroll System. The Human Resources System provides personal information about the employees. Every month, the Payroll System computes the salary of every employee, including wages, bonuses and deductions. The Mail Server and the SMS Notifier run the University e-mail and short message system services, respectively, and are used for notification purposes.

Every phone call registered by the Call Centre, except for toll-free calls, must be transformed into debits in the Payroll System. Employees can be notified by e-mail and/or short text message about their calls, so that they are aware of the deduction that shall appear in their pay slip. The only assumption we make is that the information registered in the Call Centre, apart from the data of the phone calls and the personal access codes, does not include any other information about the employees. It is the Human Resources System that provides this information.

12.3.2 Solution

The integration solution we have devised using Guaraná DSL is composed of one orchestration process that exogenously co-ordinates the applications involved in the integration solution, *cf.* Figure §12.4. Some ports use text files to communicate indirectly with the Call Centre, the Payroll System, and the SMS Notifier; we have direct access to the Human Resources System database by means of a pair of entry and exit ports. Translator tasks were used to translate messages from canonical schemas into schemas with which the integrated applications work.

The workflow begins at entry port P1, which periodically polls the Call Centre log to find new phone calls. Every phone call results in a message that is added by communicator task T1 to slot S1. The body of the message holds the polled data as stream. Every port is provided with only a single communicator task, except for entry port P1 which has a mapper task. This mapper T2

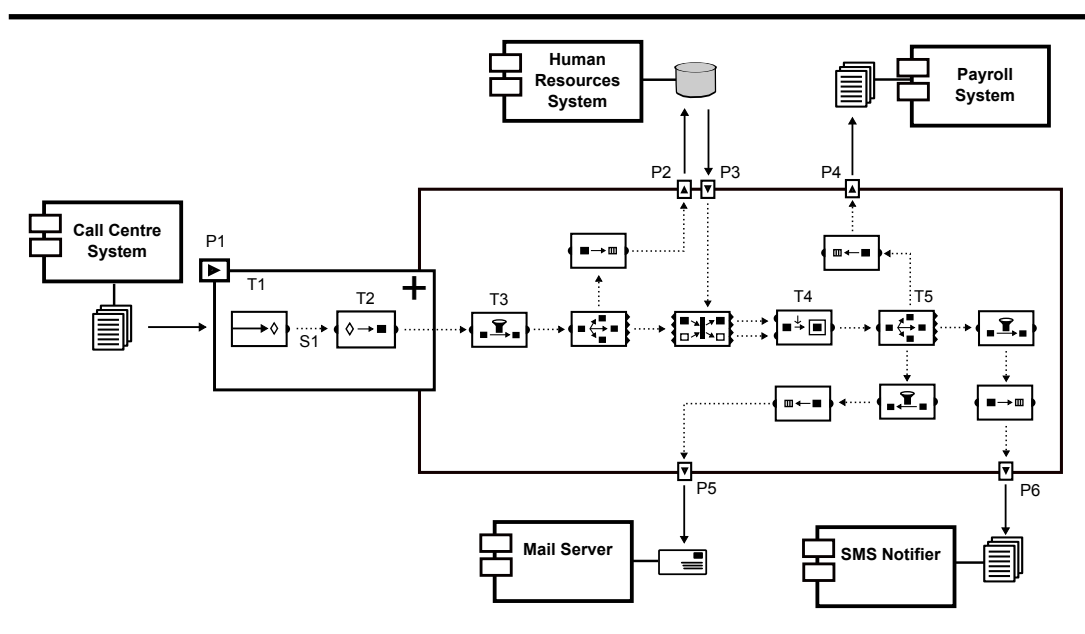


Figure 12.4: *The Unijuí University integration solution.*

maps inbound messages onto outbound messages that conform to a canonical XML schema that represents phone call records. Inside the process, task T3 filters out messages that have a toll-free call. Then, messages containing calls with a cost are replicated to the Human Resources System, so that one copy can be used to query this application, by means of ports P2 and P3. Next, task T4 enriches the other correlated copy with the information returned by the Human Resources System and then task T5 replicates this enriched message to the Payroll System, the Mail Server, and the SMS Notifier. Exit port P4 writes to the Payroll System debit orders that conform to the data model in this application. The copies sent to the Mail Server and the SMS Notifier go first through filters to prevent exit ports P5 and P6 from receiving messages without enough information (e.g., destination e-mail address and destination phone number, respectively).

12.3.3 Error detection rules

Figure §12.5 shows the rules we have specified for the Unijuí University integration solution, so that it is possible to detect errors during its execution. Rule R1 states that for every input message read at port P1, the integration

$R1 = P1[1] \longrightarrow P4[1] \ \& \ P5[?] \ \& \ P6[?]$
 $R2 = P1[1] \longrightarrow P2[0] \ \& \ P3[0] \ \& \ P4[0] \ \& \ P5[0] \ \& \ P6[0]$
 $R3 = P2[1] \longrightarrow P3[1]$

Figure 12.5: *Error detection rules for the Unijuí University solution.*

solution must produce another correlated message at port P4, and zero or one correlated messages at ports P5 and P6. Note that, if filter task (5) removes a message from the workflow in this integration solution, there shall be no correlated messages at ports P2, P3, P4, P5, and P6, causing rule R1 to fail. To avoid reporting an error in such cases, we have specified rule R2. Rule R3 states that for every request message to the Human Resources System, there must be a correlated response message.

12.3.4 Experimental results

Figure §12.6 presents the results of our experiments. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since the values we got for R^2 were 0.994, 0.996, 0.998, 0.997, and 0.998 for 1, 2, 4, 6, and 8 threads, respectively. The graph depicted for this variable shows that, as a consequence of only having four physical CPU threads available in the processor, running the integration solution with 6 and 8 threads does not help reduce the consumption of CPU time per thread very significantly, as was the case for 1, 2, and 4 threads.

The graph depicted to show the number of pending messages indicates that the integration solution supports a message production rate r until 1,200 messages per second when using 6 or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads with which we have experimented. Furthermore, these experiments indicate that for an $r = 3,000$, there is no much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. In this case study, the integration solution only starts to accumulate messages for 1 thread

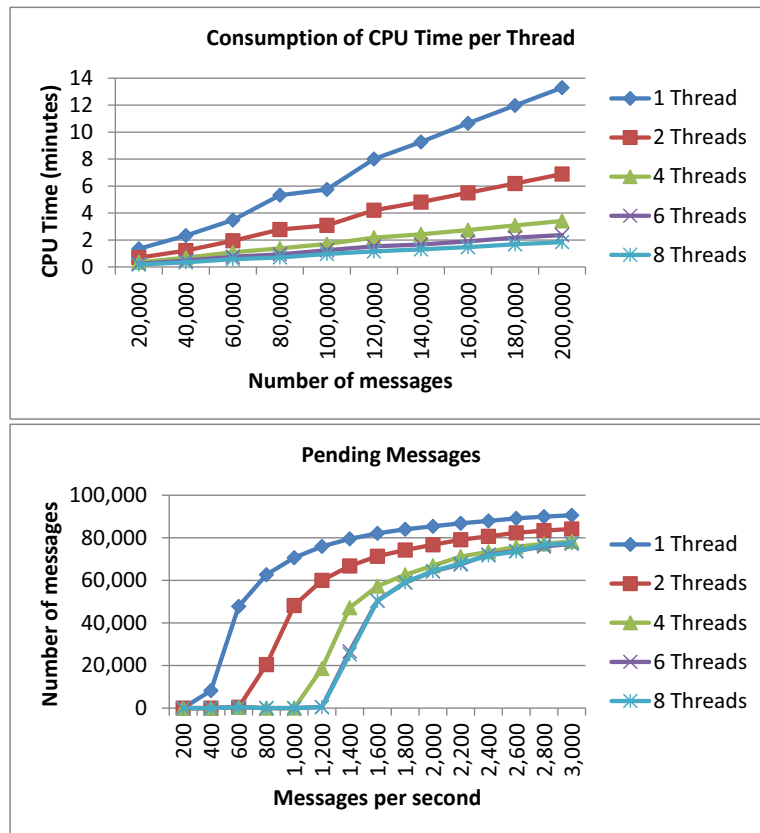


Figure 12.6: Experimental results for the Unijuí University solution.

with an $r = 400$. 2 and 4 threads allow to run the integration solution without accumulating messages with an $r = 600$ and $r = 1,000$, respectively.

12.4 Huelva's County Council

This case study consists of a real-world integration problem that builds on a project to automate the registration of new users into a unique repository of the Huelva's County Council. This repository contains information about users that comes from both a local application and a web portal. It is expected that every new user is notified and provided with his/her digital certificate by secure e-mail.

12.4.1 The software ecosystem

The integration solution involves six applications, namely: Local Users, Portal Users, LDAP, Human Resources System, Digital Certificate Platform, and Mail Server. Each application runs on a different platform, and, except for the LDAP, the Digital Certificate Platform, and the Mail Server, they were not designed with integration concerns in mind.

The Local Users is the first application developed in house; it aims to manage the county council information systems' users. Note that, this is a standalone application and does not provide an authentication service. The Portal Users is an off-the-shelf application that the web portal uses to manage its users. In addition, a unique repository for users has been set up using an LDAP-based application, so that it can provide authentication access control for several other applications inside the software ecosystem. The Human Resources System is a legacy system developed in house to provide personal information about the employees. It is a part of the integration solution since we require information like name and e-mail to compose notification e-mails. Another application developed in house is the Digital Certificate Platform, which aims to manage digital certificates; it was designed with integration concerns in mind. Amongst other services, this application can be queried to get a URL that temporarily points to a digital certificate that users can download after authenticating. Finally, the Mail Server runs the Council's e-mail service, which is used exclusively for notification purposes.

12.4.2 Solution

The integration solution we have devised using Guaraná DSL is composed of one orchestration process that exogenously co-ordinates the applications involved in the integration solution, *cf.* Figure §12.7. Some ports use text files to communicate with Local Users, Portal Users, and LDAP; the Human Resources System is queried by means of its database management system; and, the communication with the Digital Certificate Platform and the Mail Server is performed by means of APIs. Translator tasks were used to translate messages from canonical schemas into the schemas with which the integrated applications work.

The workflow begins at entry ports P1 and P2, which periodically poll the Local Users and Portal Users logs to find new users. Every port is pro-

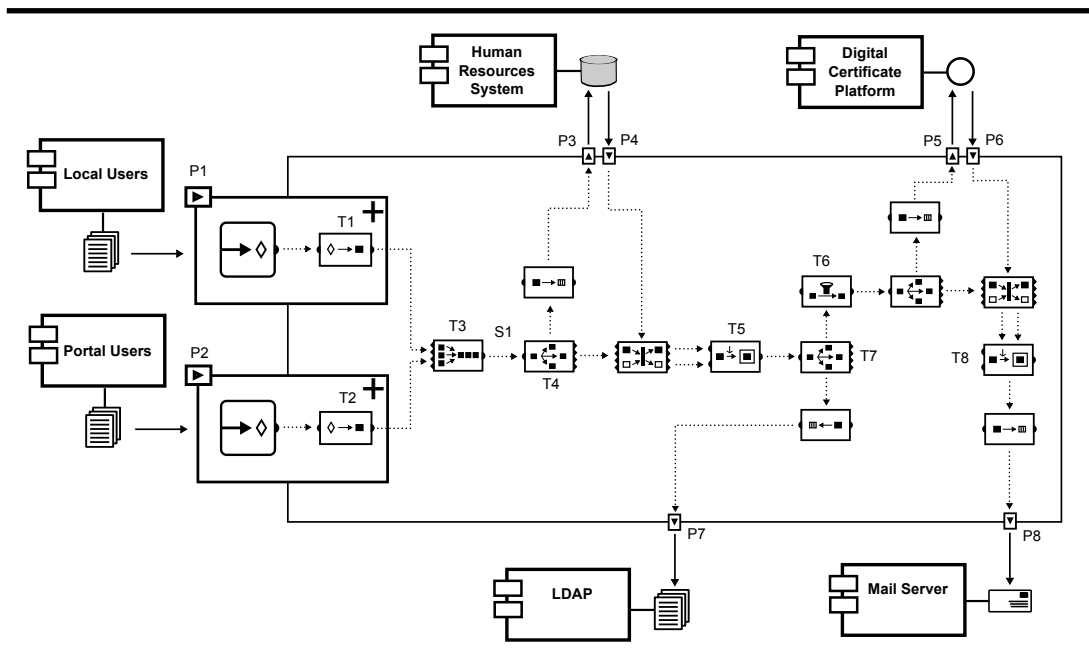


Figure 12.7: The Huelva's County Council integration solution.

vided with only a communicator task, except for ports P1 and P2 that also have a mapper task. In both ports, every user record results in a message that is added by the communicators to their corresponding slots. The body of the message holds the data that has been polled as a stream. Thus, mappers T1 and T2 map the inbound messages onto outbound messages that conform to a canonical XML schema that represents user records. Inside the process, task T3 gets messages coming from both ports and adds them to slot S1. Replicator task T4 creates two copies of every message it gets from this slot, so that one copy can be used to query application Human Resources System by means of ports P3 and P4, for information about the employee who owns a user record. Next, task T5 enriches the other correlated copy with the information returned by the Human Resources System and then task T7 replicates this enriched message with copies to the LDAP and the Digital Certificate Platform. The new user record is written to the LDAP by means of exit port P7. Before querying the Digital Certificate Platform, task T6 filters out messages that do not include an e-mail address. Messages that go through task T8, which enriches them with the corresponding certificate. Finally, exit port P8 communicates with the Mail Server application to send the certificate and notify the employee about his/her inclusion in the LDAP.

R1 = P1[1] \longrightarrow P7[1] & P8[?]
 R2 = P2[1] \longrightarrow P7[1] & P8[?]
 R3 = P3[1] \longrightarrow P4[1]
 R4 = P5[1] \longrightarrow P6[1]

Figure 12.8: Error detection rules for the Huelva's County Council solution.

12.4.3 Error detection rules

Figure §12.8 shows the rules we have specified for the Huelva's County Council integration solution. Rule R1 states that for every input message read at port P1, the integration solution must produce another correlated message at port P7, and zero or one correlated message at port P8. Rule R2 is similar to rule R1, but involves port P2. Rules R3 and R4 state that for every request message to the Human Resources System and the Digital Certificate Platform, there must be a correlated response message, respectively.

12.4.4 Experimental results

Figure §12.9 presents our experimental results. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since the values we got for the R^2 coefficient were 0.994, 0.994, 0.996, 0.996, and 0.997 for 1, 2, 4, 6, and 8 threads, respectively. The graph depicted for this variable shows that the consumption of CPU time per thread reduces considerably when adding more threads until the limit of 4 threads. This behaviour is attributed to the limit of four physical CPU threads in the processor. This explains why adding more threads to the integration solution, does not result in a significant reduction of the total CPU time per thread.

The graph depicted to show the number of pending messages, indicates that the integration solution supports a message production rate r until 800 messages per second when using 4, 6, or 8 threads, since there are not any pending messages when the message production finishes. A higher message

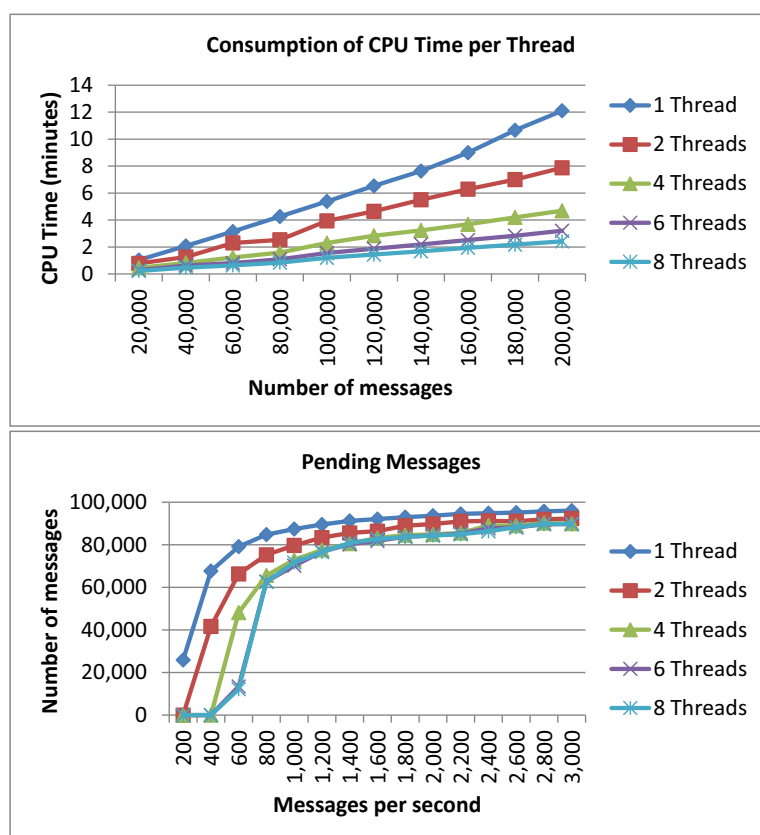


Figure 12.9: *Experimental results for the Huelva's County Council solution.*

production rate r causes the integration solution to accumulate messages, independently from the number of threads with which we have experimented. If the message production rate r ranges from 1,600 – 3,000, then there is not much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. If running the integration solution with 2 threads, no messages are accumulated until $r = 600$. A similar behaviour when using 4–8 threads can be observed in this experiment, which is attributed to the limit of four physical CPU threads in the processor.

Note that this case study and the next one are the ones that require more CPU to complete. Note that even in these cases, our proposal is able to handle a workload as high as 400 messages per second without getting collapsed.

12.5 Travel Search

Searching for flights and hotels is a well-known integration problem in the context of travel agency business systems. The goal is to devise an integration solution that makes it easier searching for the most inexpensive combination of flights and hotels for a desired travel with a limited budget, so that the customer can decide which one he/she shall book.

12.5.1 The software ecosystem

The integration solution involves two applications, namely: Flights Façade and Hotels Façade. Both applications have APIs that allows for querying several flight and hotel companies, but do not allow to restricts the results by price. Requests to the Flights Façade return information about all of the flights it can find for a specific date, departing, and destination city. Likewise, the Hotels Façade responds with all of the available hotels in a destination city for a specific date. The integration solution must provide an API so that client applications can search for flights and hotels.

12.5.2 Solution

The integration solution we have devised using Guaraná DSL is composed of one orchestration process and two wrapping processes, *cf.* Figure §12.10. The orchestration process uses a port to publish an API for searching combinations of flights and hotels. The Flights Façade and the Hotels Façade applications were provided with wrapping processes that allow to query them and remove entries that exceed the maximum affordable price for flight and hotel, respectively. We have decided to implement this functionally outside of the orchestration processes, so that it can be reused in other integration solutions.

The workflow begins at entry port P1. This port receives request messages and adds them to slot S1 inside the orchestration process, from which replicator task T1 gets them. The replicator creates two copies of every message; the first copy is used to search for flights and hotels, whereas the second is used to remove combinations of flights and hotels that exceed the total budget from the response. Task T2 promotes the request id from the body of the message to the header of the message, so that it can be used for correlation purposes in

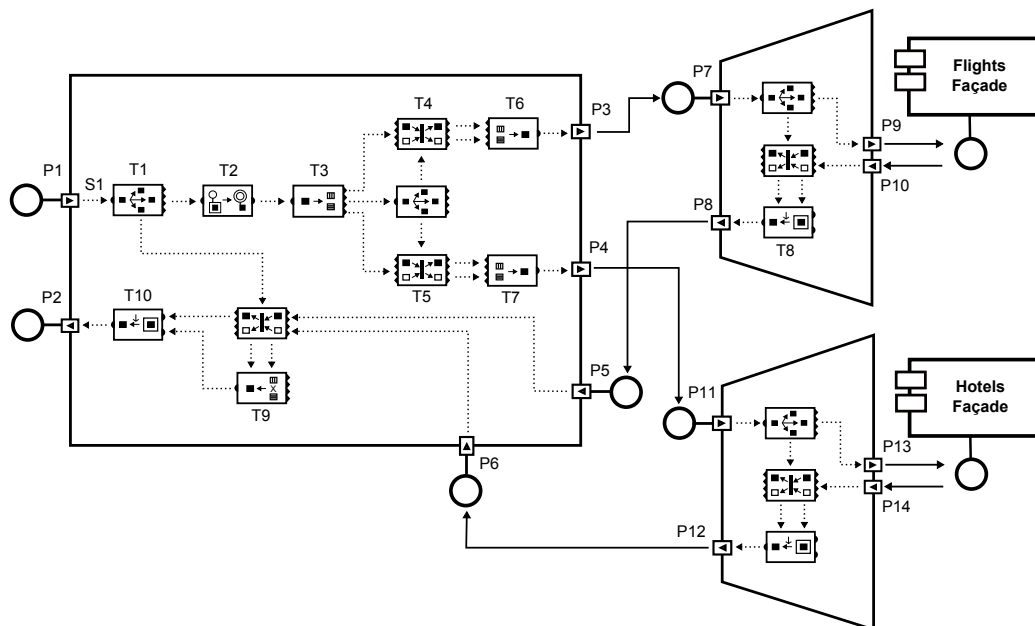


Figure 12.10: *The Travel Search integration solution.*

tasks T4 and T5. Prior to the correlation, the first copy is chopped by task T3 into different messages, so that tasks T6 and T7 can assemble the outbound messages used to request all possible flights and hotels, respectively. The wrapping process for the *Flights Façade* receives request messages from exit port P3, queries the *Flights Façade* application by means of ports P9 and P10, slims the responses in task T8, and, then, by means of exit port P8 writes the responses to the orchestration process. Symmetrically, the wrapping process for the *Hotels Façade* queries its corresponding application with messages received from exit port P4. Back to the orchestration process, task T9 builds every possible combination of flights and hotels returned by the wrapping processes, and, finally, task T10 slims the response to ensure none of the combinations exceeds the maximum budget.

12.5.3 Error detection rules

Figure §12.11 shows the rules we have specified for the Travel Search integration solution, so that it is possible to detect errors during its execution. Because this solution involves three processes, we have specified separately

Solution	Orchestration process	
R1 = P1[1] → P2[1]	R7 = P1[1] → P3[1] & P4[1]	
R2 = P1[1] → P9[1] & P13[1]		
R3 = P3[1] → P7[1]		
R4 = P8[1] → P5[1]	Flight Façade wrapper	Hotels Façade wrapper
R5 = P4[1] → P11[1]	R8 = P7[1] → P8[1]	R10 = P11[1] → P12[1]
R6 = P12[1] → P6[1]	R9 = P9[1] → P10[1]	R11 = P13[1] → P14[1]

Figure 12.11: Error detection rules for the Travel Search solution.

the rules involving two or more processes in the solution from the rules involving a single process.

Rule R1 indirectly involves all processes in the solution, since it states that for every input message read by the orchestration process at port P1, the integration solution must produce another correlated message at port P2. Rule R2 states that for every message read by the orchestration process at port P1, other two correlated messages are produced at ports P9 and P13 to query the Flights Façade and Hotels Façade, respectively. Rules R3 and R4, specify the expected behaviour for exchanging messages between the orchestration and the wrapping process for Flights Façade. The former rule states that for every message written by port P3 there must be another correlated message at port P7. The latter rule specifies the behaviour for the other way around involving ports P8 and P5. Rules R5 and R6 are similar to rules R3 and R4 for the case of the communication between the orchestration process and the wrapper.

Rule R7 states that for every input message read at port P1, the orchestration process must produce two correlated messages, one at port P3 and another at port P4, respectively.

Rule R8 is used to check the input and output of the Flights Façade, and states that for every message read at port P7, the wrapping process must produce another correlated message at port P8. Rule R9 states that for every request message to the Flights Façade, there must be a correlated response message. Rules R10 and R11 are defined for the wrapping process of Hotels Façade, and are similar to rules R8 and R9.

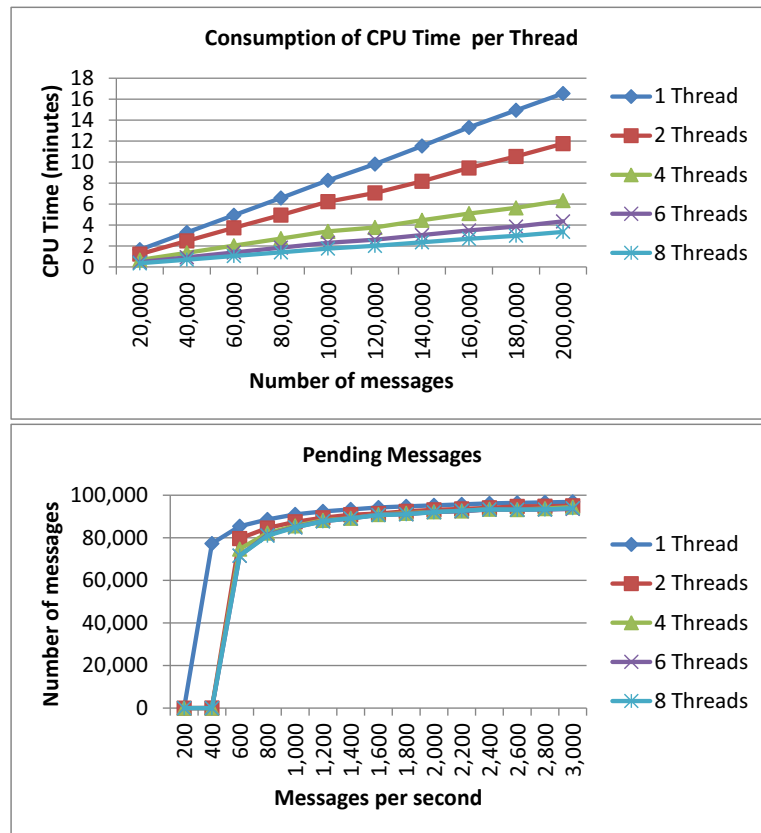


Figure 12.12: Experimental results for the Travel Search solution.

12.5.4 Experimental results

Figure §12.12 presents our experimental results. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since R^2 is 0.999, 0.998, 0.998, 0.998, and 0.999, and 0.997 for 1, 2, 4, 6, and 8 threads, respectively. Furthermore, the graph depicted for this variable shows that the consumption of CPU time per thread reduces considerably when adding more threads until the limit of 4 threads. Then, adding more threads apparently does not reduce the consumption of CPU time per thread. This behaviour is attributed to the limit of four physical CPU threads in the processor.

The graph depicted to show the number of pending messages indicate that

the integration solution supports a message production rate r until 400 messages when using 2, 4, 6 or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads. If the message production rate r ranges from 800 – 3,000, then there is not much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. The results depicted in the graph indicate for all numbers of threads t , the resulting number of pending messages is similar. This behaviour is due to the fact that the Travel Search integration solution uses more tasks that depend on two or more messages, such as the correlator, assembler, and slimmer. This has an impact on the time the integration solution requires to process a request. Furthermore, this integration solution involves three processes. The greater the number of this type of tasks and the number of processes, the more time is required to process messages.

12.6 Travel Booking

Not only requires a travel agency an integration solution that eases the process of searching for flights and hotels, but they also need to automate the booking process. Thus, the goal is to devise an integration solution that takes a travel booking request as input and books the flights and the hotel specified.

12.6.1 The software ecosystem

The integration solution involves five applications, namely: Travel System, Invoice System, Mail Server, Flights Façade, and Hotels Façade. The Travel System is an off-the-shelf software system that the travel agency uses to register information about their customers and booking requests. The invoice service runs on the Invoice System, which is a separate software system that allows customers to pay their travels using their credit cards. The Mail Server runs the e-mail service and is used for providing customers with information about their bookings. The Flights Façade and the Hotels Façade represent interfaces that allow booking flights and hotels. They both, in addition to the Mail Server, represent applications that were designed with integration concerns in mind. Contrarily, the Travel System and the Invoice System are software systems that were designed without taking integration into account,

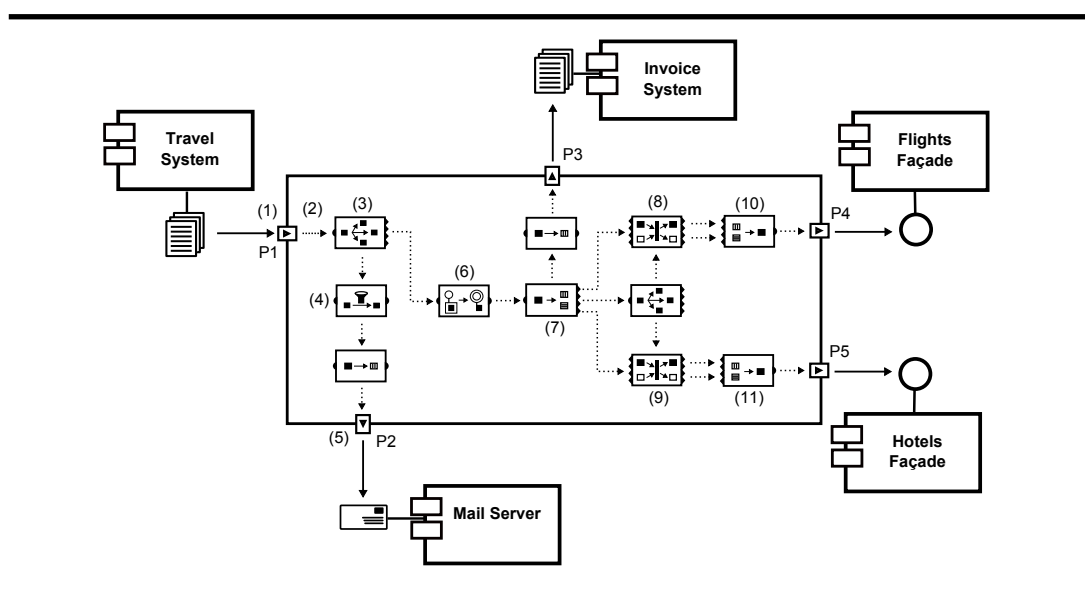


Figure 12.13: *The Travel Booking integration solution.*

thus, the integration solution must interact with them by means of their data layer. The only assumption we make is that every booking registered in the Travel System contains all of the necessary information about the payment, flight and hotel, and a record locator which uniquely identifies the booking.

The integration solution must periodically poll the Travel System for new travel bookings, so that flights and hotel can be booked, the customer can be invoiced and provided with a piece of e-mail with the information about his/her travels.

12.6.2 Solution

The integration solution we have devised using Guaraná DSL is composed of one orchestration process that exogenously co-ordinates the applications involved in the integration solution, *cf.* Figure [§12.13](#). Some ports have access to the Travel System and the Invoice System data layers by means of files. Translator tasks were used in the process to translate messages from canonical schemas into the schemas with which the integrated applications Invoice System and Mail Server work.

The workflow begins at entry port P1, which periodically polls the Travel

$$R1 = P1[1] \longrightarrow P2[?] \ \& \ P3[1] \ \& \ P4[1] \ \& \ P5[1]$$

Figure 12.14: *Error detection rules for the Travel Booking solution.*

System to find new bookings. Bookings are stored in individual XML files. For every booking, the entry port inputs a message to the process, which is in turn added to slot S1. Task T1 gets messages from this slot and replicates them, so that one copy is used to send the e-mail to the customer and the other is used to prepare the invoice and the booking. The first copy goes through filter task T2, which prevents exit port P2 from receiving messages without a destination e-mail address. Task T3 promotes the record locator from the body of the message to the header of the message, so that it can be used for correlation purposes in tasks T5 and T6. Prior to the correlation, the second copy is chopped by task T4 into different messages, so that one outbound message with the payment information goes to the *Invoice System* and tasks T7 and T8 can assemble the messages used to book the flights and the hotel, respectively.

12.6.3 Error detection rules

Figure §12.14 shows the rules we have specified for the Travel Book integration solution, so that it is possible to detect errors during its execution.

The single rule specified for this solution states that for every input message read at port P1, the integration solution must produce zero or one correlated message at port P2, and another correlated message at ports P3, P4, and P5.

12.6.4 Experimental results

Figure §12.15 presents our experimental results. The consumption of CPU time grows linearly as the number of messages m increases, independently from the number of threads t available. We performed a linear regression analysis and confirmed the previous claim since R^2 is 1, 0.999, 0.998, 0.999, and 0.998 for 1, 2, 4, 6, and 8 threads, respectively. A great reduction in the consumption of CPU time is achieved when adding more threads until the limit of 4 threads. Then, adding more threads apparently does not reduce the

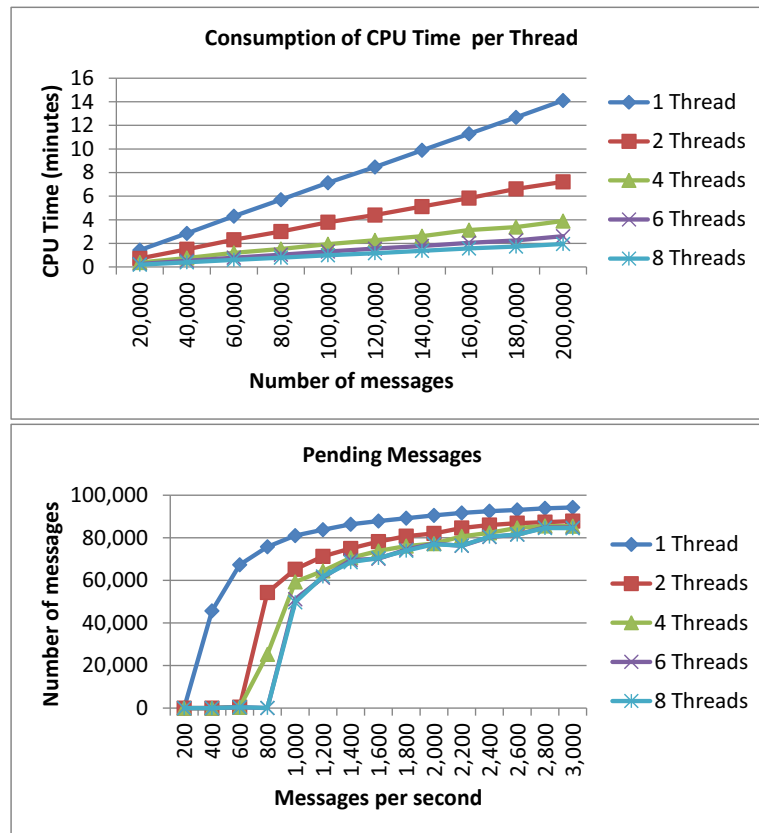


Figure 12.15: *Experimental results for the Travel Booking solution.*

consumption of CPU time per thread, because the CPU has only four physical threads available.

The experimental results that we obtained for the number of pending messages, indicates that this integration solution supports a message production rate r until 800 messages when using 6 or 8 threads, since there are not any pending messages when the message production finishes. A higher message production rate r causes the integration solution to accumulate messages, independently from the number of threads. Furthermore, these experiments indicate that for a very high message production rate r of 3,000 there is no much difference in using 1 or 8 threads. With $r = 200$, messages do not accumulate even if running the integration solution with only 1 thread. If running the integration solution with 2 and 4 threads, no messages are accumulated until $r = 600$.

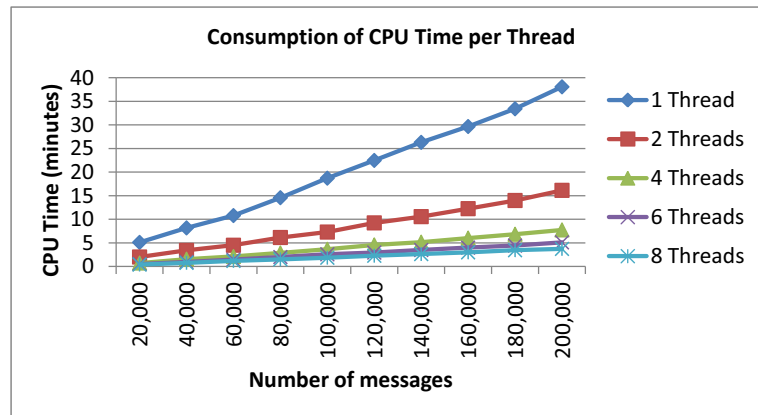


Figure 12.16: Experimental results for the Café solution using JBI adapters.

12.7 An experiment using JBI adapters

As we mentioned in this dissertation at Chapter §8, our Domain-Specific Language, Software Development Kit, and Runtime System allow to reuse the catalogue of binding components provided by Open ESB. On one hand, this is an advantage because we could design and implement our proposal without having to spend much time at designing and implementing our own catalogue of adapters. This approach has also helped us to concentrate efforts on the core architecture of our proposal, since the adapters are peripheral to its main architecture. On the other hand, our experiments have shown that it is very expensive to design and implement integration solutions using JBI adapters. Furthermore, the CPU time increases much because we have to run the Open ESB server, which is responsible for managing the binding components with which our JBI adapters have to communicate through the network.

In this section, we present the CPU times consumed by an implementation of the Café solution using JBI adapters. The times were measured using the memory monitor available in our Runtime System. This monitor allows to gather the CPU time consumed by an integration solution running in our Runtime System, however the monitor cannot measure the CPU time consumed by Open ESB to run the binding components. The graph depicted in Figure §12.16, shows the results measured for the consumption of CPU time running the Café solution using JBI adapters. The consumption of CPU time

grows linearly as the number of messages m increases, independently from the number of threads t available. A similar behaviour was observed when we ran the Café solution with mock adapters, *cf.* Section §12.2. The linear growth is confirmed by a linear regression analysis we performed and the R^2 coefficients we calculated for 1, 2, 4, 6, and 8 threads, which were 0.997, 0.994, 0.998, 0.998, and 0.999, respectively. The consumption of CPU time per thread reduces significantly until 4 threads, then the results indicate the same behaviour as for the experiments in the other case studies regarding variable to measure the consumption of CPU time.

12.8 Summary

In this chapter, we have reported on five case studies that introduce different integration problems. For each of them, we have designed an integration solution using our Domain-Specific Language. Every integration solution also includes a number of rules to detect possible errors during its execution. Furthermore, we have implemented every integration solution using our Software Development Kit, and have executed several experiments to evaluate their execution using our Runtime System in case studies with high workload.

Part IV

Final Remarks

Chapter 13

Conclusions

*If you can look into the seeds of time,
And say which grain will grow and which will not;
Speak then to me.*

William Shakespeare, British author (1564-1616)

Enterprise Application Integration is widely-used by companies that aim at reusing the applications that are available within their software ecosystems to support and optimise their business processes. The catalogue of integration patterns proposed by Hohpe and Woolf [54] was adopted by the Enterprise Application Integration community as a cookbook to design and implement integration solutions. As of the time of writing this dissertation, there are not many domain-specific software tools to help software engineers devise Enterprise Application Integration solutions based on integration patterns. In this dissertation we have analysed Camel, Spring Integration, and Mule, which are three successful open-source tools. Companies that provide Enterprise Application Integration solutions are interested in software tools that can be easily adapted to focus on specific contexts such as e-commerce, health systems, financial systems, and insurance systems, in which they have to meet standards and recommendations like RosettaNet [96], HL7 [52], SWIFT [106], and HIPAA [51], respectively.

We have used fifteen of the measures proposed by Chidamber and Kemerer [16], Henderson-Sellers [50], Martin [73], and McCabe [74] to evaluate the maintainability of Camel, Spring Integration, and Mule. The results that

we obtained confirm our hypothesis that building on these software tools for particular contexts may be costly. Throughout this dissertation, we have reported on Guaraná, which is our proposal to design and implement Enterprise Application Integration solutions. We have used the fifteen maintainability measures to evaluate Guaraná and compare it to Camel, Spring Integration, and Mule. The analysis of the results supports our thesis by indicating that Guaraná is much easier to maintain, and, consequently to adapt for specific contexts. We have also developed five case studies to demonstrate that Guaraná is viable to be used in real-world integration problems.

We have developed Guaraná in the context of the Model-Driven Engineering discipline. This discipline allowed us to make models first-class citizens in the development process of integration solutions. A set of transformations were devised to allow the automatic translation of models into code. Not only helps this approach to reduce the time to develop an integration solution, but also helps to isolate models from their implementation and improve the documentation of integration solutions.

Our Domain-Specific Language, which is referred to as Guaraná DSL, offers a graphical notation that can be used to represent the models of integration solutions with a high-level of abstraction. Guaraná DSL defines a set of constructs for the core abstractions involved in the design of integration solutions, and a general-purpose toolkit which enhances our language with other constructs to support most of the integration patterns by Hohpe and Woolf [54]. Guaraná DSL can also be used by software engineers as a common and yet simple vocabulary for communicating in this field. Ports are one of the abstractions provided by Guaraná DSL. They are used by integration solutions to interact with the applications being integrated. The current version of Guaraná DSL includes one-way ports: entry and exit. In a scenario in which the integration solution makes a request and waits for a response, the correlation of the request and the response messages must be accomplished by using other constructs provided by the general-purpose toolkit. The same happens in a scenario in which the integration solution exports an interface to receive requests and produce responses. Throughout the development of this dissertation, we have noticed that it would be interesting to provide two-way ports, which internally do the required correlation automatically.

The Software Development Kit, which is referred to as Guaraná SDK, is our implementation for the abstractions defined in our Domain-Specific Language. Guaraná SDK is a software tool that can be used to implement integration solutions. It is a Java-based command query API [31]. By using our trans-

formation scripts, software engineers can automatically translate their models into Guaraná SDK. The architecture of Guaraná SDK is organised into two layers, namely: framework and toolkit. The former provides a number of classes and interfaces that implement the abstractions of our Domain-Specific Language, whereas the latter extends some abstractions in the former to provide concrete adapters and tasks that support several integration patterns. The framework layer includes a Runtime System, which allows to execute integration solutions. The execution model of our Runtime System is totally asynchronous. Supporting transactions in this kind of execution models requires more research effort. Tasks that comprise the integration workflow of integration solutions, notify they are ready to be executed when they have messages available in all their inputs. For each execution of a task, our Runtime System generates a work unit and adds it to a work queue. There can be several threads that listen to this queue and are responsible for their actual execution. We have noticed that the Runtime System cannot guarantee that the work units are executed homogeneously in scenarios with a very high workload. The reason is that there can be many more work units for the first task, than for the second one, than for the third one, and so on. Consequently, the time required by an integration solution to process an inbound message increases very much. It is necessary to study this problem in depth, and we believe that some components in the current Runtime System could be replaced by the scheduling mechanism provided by Java 5.0.

We have developed a monitoring system that aims to detect possible errors during the execution of an integration solution. Our failure semantics include errors when reading from or writing to a resource, structural, and deadline errors. To evaluate this system, we have executed several experiments involving six well-known patterns that lie at the core of most real-world integration solutions, namely: pipeline, dispatcher, merger, request-reply, splitter, and aggregator [54]. The current version of the monitoring system does not detect errors that can happen inside a process during the processing of a message. We have found that this is a very frequent and thus an important error that has to be included into our failure semantics. Furthermore, our monitoring system should be enhanced with the capacity to observe slots inside an integration solution, so it can detect messages that get stuck in a slot and probably shall never be processed. A message can get stuck in situations in which it needs to be processed together with a correlated message, which does not arrive due to some unexpected failure. Thus another kind of failure shall be included into our failure semantics to tackle such situations. Regarding the use of rules to help in the process of error detection, we have noticed that software engineers have to be careful when an integration solution uses filters, since they may

remove a message from the workflow; we have noticed that it is not difficult to misinterpret the semantics of our rules in such cases.

We have applied Guaraná to five case studies carried out. In our experiments, we measured two variables: the consumption of CPU time per thread and the number of pending messages. The results indicate that Guaraná is viable and can be used to solve real-world integration problems. Every case study was modelled using Guaraná DSL and implemented using Guaraná SDK. We also provide the rules for the detection of errors in these solutions. Guaraná was designed to be integrated with Open ESB [90], so that we can reuse its catalogue of binding components. The binding components implement the low-level transport protocol required to interact with applications. There are many types of binding components available nowadays, which allow integration solutions to connect to almost every existing application in a software ecosystem. Binding components are part of the JBI specification. Throughout the development of this dissertation we have noticed that the use of JBI adapters results in a very high consumption of CPU Time. The use of JBI adapters does not have an impact on the number of pending messages, since once a message is read into the integration solution its processing does not depend on the Open ESB anymore, but on our Runtime System. As a conclusion, we believe that future versions of Guaraná should provide their own adapters, so that the consumption CPU time can be reduced.

During the development of this dissertation we carried out several collaborations with other international and Spanish research groups. Three research visits were paid to the Newcastle University (United Kingdom), the University of Leicester (United Kingdom), and the Polytechnic Institute of Leiria (Portugal). In collaboration with the Newcastle University, we worked on the field of fault-tolerance; at the University of Leicester, we presented our research results and gathered feedback from a group of researchers working on Domain-Specific Languages; at the Polytechnic Institute of Leiria, we presented our research results and worked on applying techniques for optimisation to the Runtime System of Guaraná. We have also collaborated with a research group from the Federal University of Rio Grande do Sul (Brazil), with which two workshops were organised to present research results and to prepare a proposal for a joint research project. With the Slovak University of Technology in Bratislava (Slovakia), we collaborated in the context of a master thesis that has delved into how to find and automatically generate optimal integration solutions. In Spain, we had three collaborations at the University of Seville, in which two workbenches to support Guaraná were developed, and a runtime based on Microsoft Windows Workflow Foundation to execute in-

tegration solutions designed with Guaraná was developed as well. Last, but not least, we collaborated with two Spanish companies and a public administration. In collaboration with the Intelligent Dialogue Systems, S.L. (Spain), we applied partial results on our Domain-Specific Language to real-world problems. We have collaborated with the Intelligent Integration Factory, S.L. (Spain) to validate and transfer the research results in this dissertation to the industry. At the Huelva's County Council we have collaborated to apply partial results of Guaraná to solve real-world problems and gather feedback.

Summing up, we have managed to devise a proposal for Enterprise Application Integration that has proven to be useful and viable, and we have started a number of collaborations that we hope shall help Guaraná find its way into the industry.

Bibliography

- [1] A. Álvarez. *A proof of concept for Guaraná DSL*. Report, Huelva's County Council, 2011
- [2] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. *Basic concepts and taxonomy of dependable and secure computing*. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.
- [3] M. Azanza, O. Díaz, and S. Trujillo. *Software Factories: Describing the assembly process*. In *ICSP*, 2010.
- [4] P. Baker, S. Loh, and F. Weil. *Model-driven engineering in a large industrial context: Motorola case study*. In *MoDELS*, pages 476–491, 2005.
- [5] T. G. Baker. *Lessons learned integrating COTS into systems*. In *ICCBSS*, pages 21–30, 2002.
- [6] L. D. Balk and A. Kedia. *PPT: a COTS integration case study*. In *ICSE*, pages 42–49, 2000.
- [7] C. Ballard, A. Gupta, V. Krishnan, N. Pessoa, and O. Stephan. *Data mart consolidation: Getting control of your enterprise information*. IBM Press, 2005.
- [8] S. Bergin and J. Keating. *A case study on the adaptive maintenance of an Internet application*. *Journal of Software Maintenance*, 15(4):254–264, 2003.
- [9] J. Bézivin. *On the unification power of models*. *Software and System Modeling*, 4(2):171–188, 2005.
- [10] J. Bézivin, S. Hammoudi, D. Lopes, and F. Jouault. *Applying MDA approach for web service platform*. In *Enterprise Distributed Object Computing Conference*, pages 58–70, 2004.

- [11] P. Bradáč. *Model-driven enterprise application integration*. Master's thesis, Slovak University of Technology in Bratislava, 2011
- [12] G. S. Brodal. *Worst-case efficient priority queues*. In *7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 52–58, 1996.
- [13] B. Bukovics. *Pro WF: Windows Workflow in .NET 3.0*. Apress, 2007
- [14] R. H. Campbell and B. Randell. *Error recovery in asynchronous systems*. *IEEE Trans. Software Eng.*, 12(8):811–826, 1986.
- [15] D. Chappel. *Enterprise Service Bus: Theory in practice*. O'Reilly, 2004
- [16] S. R. Chidamber and C. F. Kemerer. *A metrics suite for object-oriented design*. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.
- [17] B. A. Christudas. *Service-oriented Java business integration*. Packt, 2008
- [18] P. Clements and L. Northrop. *Software product lines: Practices and patterns*. Addison-Wesley, 2001
- [19] S. Cook. *Domain-specific modeling and model driven architecture*. *MDA Journal*, pages 2–10, 2004.
- [20] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-specific development with Visual Studio DSL tools*. Addison-Wesley, 2007
- [21] R. Corchuelo, J. L. Arjona, D. Ruiz, J. L. Álvarez, R. Z. Frantz, C. Molina-Jiménez, I. Hernández, C. R. Osuna, A. M. Reina-Quintero, H. Sleiman, I. Fernández, and P. Jiménez. *A roadmap on integrating applications and data on the Web*. In *Jornadas de Ingeniería del Software y Bases de Datos*, pages 133–142, 2010.
- [22] R. Corchuelo, R. Z. Frantz, and J. González. *Una comparación de ESBs desde la perspectiva de la integración de aplicaciones*. In *Jornadas de Ingeniería del Software y Bases de Datos*, pages 403–408, 2008.
- [23] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. *Learning from project history: a case study for software development*. In *Conference on Computer Supported Cooperative Work*, pages 82–91, 2004.
- [24] J. Davies, D. Schorow, S. Ray, and D. Rieber. *The definitive guide to SOA: Enterprise Service Bus*. Apress, 2008

- [25] B. Demuth. *The Dresden OCL toolkit and its role in information systems development*. In *Int. Conf. on Information Systems Development*, pages 1–12, 2004.
- [26] X. Dong and M. W. Godfrey. *Understanding source package organization using the hybrid model*. In *International Conference on Software Maintenance*, 2009.
- [27] D. Dossot and J. D’Emic. *Mule in action*. Manning, 2009
- [28] A. Epping and C. M. Lott. *Does software design complexity affect maintenance effort?* In *NASA/Goddard 19th Annual Software Engineering Workshop*, pages 297–313, 1994
- [29] J. Estublier, G. Vega, and A. D. Ionita. *Composing domain-specific languages for wide-scope software engineering applications*. In *MoDELS*, pages 69–83, 2005.
- [30] M. Fisher, J. Partner, M. Bogoevici, and I. Fuld. *Spring Integration in action*. Manning, 2010
- [31] M. Fowler. *Domain-specific languages*. Addison-Wesley, 2010
- [32] R. B. France and J. M. Bieman. *Multi-view software evolution: A UML-based framework for evolving object-oriented software*. In *ICSM*, pages 386–395, 2001.
- [33] R. Z. Frantz. *A DSL for enterprise application integration*. *International Journal of Computer Applications in Technology*, 33(4):257–263, 2008.
- [34] R. Z. Frantz and R. Corchuelo. *A software development kit to implement integration solutions*. In *27th Symposium On Applied Computing*, 2012 (To be published).
- [35] R. Z. Frantz, R. Corchuelo, and J. L. Arjona. *An efficient orchestration engine for the Cloud*. In *IEEE International Conference on Cloud Computing Technology and Science*, pages 711–716, 2011.
- [36] R. Z. Frantz, R. Corchuelo, and J. González. *Advances in a DSL for application integration*. In *ZOCO*, pages 54–66, 2008.
- [37] R. Z. Frantz, R. Corchuelo, and C. Molina-Jiménez. *Towards a fault-tolerant architecture for enterprise application integration solutions*. In *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 294–303, 2009.

- [38] R. Z. Frantz, R. Corchuelo, and C. Molina-Jiménez. *An architecture to design enterprise application integration solutions with fault tolerance support*. In *VI Jornadas Científico-Técnicas en Servicios Web y SOA*, pages 51–62, 2010.
- [39] R. Z. Frantz, R. Corchuelo, and C. Molina-Jiménez. *Error-detection in enterprise application integration solutions*. In *Conference on ENTERprise Information Systems*, pages 170–179, 2011.
- [40] R. Z. Frantz, R. Corchuelo, and C. Molina-Jiménez. *A fault-tolerance mechanism to detect errors in enterprise application integration solutions*. *International Journal of Systems and Software*, 2011 (To be published).
- [41] R. Z. Frantz, R. Corchuelo, C. R. Osuna, and C. Molina-Jiménez. *Monitoring errors in integration workflows*. In *International Conference on Software Engineering Research and Practice*, pages 598–604, 2011.
- [42] R. Z. Frantz, C. Molina-Jimenez, and R. Corchuelo. *On the design of a domain specific language for enterprise application integration solutions*. In *International Workshop on Model-Driven Engineering*, pages 19–30, 2010.
- [43] R. Z. Frantz, A. M. Reina-Quintero, and R. Corchuelo. *A Domain-Specific language to design enterprise application integration solutions*. *International Journal of Cooperative Information Systems*, 20(2):143–176, 2011.
- [44] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1994
- [45] D. Ghosh. *DSLs in action*. Manning Publications Co., 2011
- [46] G. Giachetti, B. Marín, and Ó. Pastor. *Using UML as a domain-specific modeling language: A proposal for automatic generation of UML profiles*. In *Conference on Advanced Information Systems Engineering*, pages 110–124, 2009.
- [47] J. Greenfield and K. Short. *Software Factories: Assembling applications with patterns, models, frameworks and tools*. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, 2003.
- [48] J. L. Gross and J. Yellen. *Handbook of graph theory*. CRC Press, 2003

- [49] B. Hailpern and P. L. Tarr. *Model-driven development: The good, the bad, and the ugly*. *IBM Systems Journal*, 45(3):451–462, 2006.
- [50] B. Henderson-Sellers. *Object-oriented metrics, measures of complexity*. Prentice Hall, 1996
- [51] *Health insurance portability and accountability act home*, 2011.
- [52] *Health level seven international home*, 2011.
- [53] G. Hohpe. *Your coffee shop doesn't use two-phase commit*. *IEEE Software*, 22(2):64–66, 2005.
- [54] G. Hohpe and B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, 2003
- [55] J. E. Hopcroft and R. E. Tarjan. *Efficient algorithms for graph manipulation*. *Communications of the ACM*, 16(6):372–378, 1973.
- [56] V. Hoyer, F. Gilles, T. Janner, and K. Stanoevska-Slabeva. *SAP research RoofTop marketplace: Putting a face on service-oriented architectures*. In *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 2009.
- [57] *IBM mashup center home*, 2011.
- [58] C. Ibsen and J. Anstey. *Camel in action*. Manning, 2010
- [59] IEEE. *IEEE standard glossary of software engineering terminology*. IEEE Computer Society, 1990.
- [60] *Jackbe Presto home*, 2011.
- [61] M. Jorgensen. *An empirical study of software maintenance tasks*. *Journal of Software Maintenance*, 7(1):27–48, 1995.
- [62] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. *ATL: A model transformation tool*. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.
- [63] S. Kent. *Model driven engineering*. In *Integrated Formal Methods*, pages 286–298, 2002.
- [64] T. Koponen. *Evaluation framework for open source software maintenance*. In *International Conference on Software Engineering Advances*, page 52, 2006.

- [65] J. Kovse and T. Härder. *Generic XMI-Based UML model transformations*. In *Object Oriented Information Systems*, pages 192–198, 2002.
- [66] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. *Model-based DSL frameworks*. In *OOPSLA Companion*, pages 602–616, 2006.
- [67] G. Lenz and C. Wienands. *Practical Software Factories in .NET*. Apress, 2006
- [68] J. Li, R. Conradi, C. Bunse, M. Torchiano, O. P. N. Slyngstad, and M. Morisio. *Development with off-the-shelf components: 10 facts*. *IEEE Software*, 26(2):80–87, 2009.
- [69] P. F. Linington. *Automating support for e-business contracts*. *Int. J. Cooperative Inf. Syst.*, 14(2-3):77–98, 2005.
- [70] D. G. Lobato. *Graphical editor and code generator for Guaraná DSL 1.2*. Report, University of Seville, ETSI Informática, 2011
- [71] M. Lorenz and J. Kidd. *Object oriented software metrics*. Prentice Hall, 1994
- [72] G. D. Lorenzo, H. Hacid, H.-Y. Paik, and B. Benatallah. *Data integration in mashups*. *SIGMOD Record*, 38(1):59–66, 2009.
- [73] R. C. Martin. *Agile software development, principles, patterns, and practices*. Prentice Hall, 2002
- [74] T. J. McCabe. *A complexity measure*. *IEEE Trans. Software Eng.*, 2(4): 308–320, 1976.
- [75] P. Mederly and P. Návrat. *Construction of messaging-based integration solutions using constraint programming*. In *ADBIS*, pages 579–582. Springer, 2010.
- [76] P. Mederly and P. Návrat. *Automated design of integration solutions based on messaging*. In *DATAKON*, pages 1–10. Springer, 2011
- [77] T. Mens and P. V. Gorp. *A taxonomy of model transformation*. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
- [78] D. Messerschmitt and C. Szyperski. *Software ecosystemm: Understanding an indispensable technology and industry*. MIT Press, 2003
- [79] J. Miller and J. Mukerji. *MDA guide version 1.0.1*, 2003.

- [80] [Meta object facility specification version 2.0](#), 2006.
- [81] [MOF model to text transformation language version 1.0](#), 2008.
- [82] J. Muñoz and V. Pelechano. [MDA versus Software Factories](#). In *DSDM*, 2005.
- [83] J. Offutt, A. Abdurazik, and S. R. Schach. [Quantitatively measuring object-oriented couplings](#). *Software Quality Journal*, 2008.
- [84] J. Oldevik, T. Neple, R. Gronmo, J. Oyvind Aagedal, and A.-J. Berre. [Toward standardised model to text transformations](#). In *European Conference on Model Driven Architecture*, pages 239–253, 2005.
- [85] OMG. [Meta object facility version 2.0 query/view/transformation specification](#). Technical report, OMG, 2005.
- [86] [UML 2.3 superstructure specification](#), 2010.
- [87] [Open ESB home](#), 2010.
- [88] T. Pfarr and J. E. Reis. [The integration of COTS/GOTS within NASA's HST command and control system](#). In *ICCBSS*, pages 209–221, 2002.
- [89] P. Ponniah. *Data warehousing fundamentals for IT professionals*. Wiley, 2010
- [90] T. Rademakers and J. Dirksen. *Open-source ESBs in action*. Manning, 2009
- [91] N. Randolph, D. Gardner, C. Anderson, and M. Minutillo. *Professional Visual Studio 2010*. Wrox, 2010
- [92] A. Redkar, C. Walzer, S. Boyd, R. Costall, K. Rabold, and T. Redkar. *Pro MSMQ: Microsoft Message Queue Programming*. Apress, 2004
- [93] A. W. S. Regalado. *Domain-specific languages*. Master's thesis, ETSI Informática, 2008
- [94] C. Renouf. *Pro IBM WebSphere application server 7 internals*. Apress, 2009
- [95] M. Richards, R. Monson-Haefel, and D. A. Chappell. *Java Message Service*. O'Reilly, 2009
- [96] [RosettaNet home](#), 2011.

- [97] I. Samoladas, I. Stamelos, L. Angelis, and A. Oikonomou. [Open source software development should strive for even greater code maintainability](#). *Commun. ACM*, 47(10):83–87, 2004.
- [98] G. Samtani. *B2B integration: A practical guide to collaborative e-commerce*. Imperial College Press, 2003
- [99] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt. [Maintainability of the Linux kernel](#). *IEE Proceedings: Software*, 149(1):18–23, 2002.
- [100] D. C. Schmidt. [Guest editor’s introduction: Model-driven engineering](#). *IEEE Computer*, 39(2):25–31, 2006.
- [101] N. F. Schneidewind. [The state of software maintenance](#). *IEEE Trans. Software Eng.*, 13(3):303–310, 1987.
- [102] H. A. Sleiman, A. W. S. Regalado, R. Z. Frantz, and R. Corchuelo. [Towards automatic code generation for EAI solutions using DSL tools](#). In *Jornadas de Ingeniería del Software y Bases de Datos*, pages 134–145, 2009.
- [103] H. A. Sleiman. *Web application integration: An approach based on dsl and workflow*. Master’s thesis, ETSI Informática, 2008
- [104] G. P. Souza and C. F. R. Geyer. *Tuplebiz: Distributed tuple space with byzantine fault-tolerance support*. Technical report 365, Federal University of Rio Grande do Sul, 2011
- [105] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse modeling framework*. Addison-Wesley, 2008
- [106] [Society for worldwide interbank financial telecommunication home](#), 2011.
- [107] J.-P. Tolvanen and S. Kelly. [Defining domain-specific modeling languages to automate product derivation: Collected experiences](#). In *SPLC*, pages 198–209, 2005.
- [108] A. Vallecillo. [A journey through the secret life of models](#). In *Perspectives Workshop: Model Engineering of Complex Systems (MECS)*, pages 1–23, 2008.
- [109] A. van Deursen and P. Klint. [Little languages: Little maintenance?](#) *Journal of Software Maintenance*, 10(2):75–92, 1998.

- [110] F. B. Vernadat. *Enterprise integration and interoperability*. In *Springer Handbook of Automation*, pages 1529–1538. Springer, 2009.
- [111] J. Weiss. *Aligning relationships: Optimizing the value of strategic outsourcing*. Technical report, IBM, 2005.
- [112] *WSO2 mashup server home*, 2011.
- [113] *Yahoo! Pipes home*, 2011.
- [114] L. Yu. *Indirectly predicting the maintenance effort of open-source software*. *Journal of Software Maintenance*, 18(5):311–332, 2006.
- [115] L. Yu. *Common coupling as a measure of reuse effort in kernel-based software with case studies on the creation of MkLinux and Darwin*. *Journal of the Brazilian Computer Society*, 14:45–55, 2008.
- [116] L. Yu, S. R. Schach, and K. Chen. *Measuring the maintainability of open-source software*. In *ISESE*, pages 297–303, 2005.
- [117] L. Yu, S. R. Schach, K. Chen, G. Z. Heller, and A. J. Offutt. *Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD, NetBSD, and OpenBSD*. *Journal of Systems and Software*, 79(6):807–815, 2006.
- [118] L. Yu, S. R. Schach, K. Chen, and A. J. Offutt. *Categorization of common coupling and its application to the maintainability of the Linux kernel*. *IEEE Trans. Software Eng.*, 30(10):694–706, 2004.

This document was typeset on 2012/2/3 at 12:38 using class `RC-BOK` α 2.12 for `LATEX2 ϵ` . As of the time of writing this document, this class is not publicly available since it is in alpha version. Only members of The Distributed Group are using it to typeset their documents. Should you be interested in giving forthcoming public versions a try, please, do contact us at contact@tdg-seville.info

*T*ypical companies rely on their software ecosystems to support and optimise their business processes. Software ecosystems are composed of many applications that were not usually designed taking integration into account. Enterprise Application Integration provides methodologies and tools to design and implement integration solutions.

The Enterprise Application Integration community has adopted the catalogue of integration patterns proposed by Hohpe and Woolf as a cookbook to design and implement integration solutions. Furthermore, there are a few software tools to help software engineers devise enterprise application integration solutions that are based on integration patterns. Some companies are interested in adapting these software tools to support their domain-specific tools to specific contexts.

In this dissertation, our research hypothesis is that the current software tools are not so easy to maintain as expected, thus increasing the costs of these adaptation process. Our goal in this dissertation is to support the thesis that it is possible to devise a domain-specific language and a set of domain-specific tools to design and implement Enterprise Application Integration solutions that are easier to maintain than the current software tools. Our core contribution consists of a Domain-Specific Language that software engineers can use to represent the models they design for their integration problems at a high-level of abstraction; a Software Development Kit that can be used to implement and run integration solutions; transformations that allow for the automatic translation of models into code; and a monitoring system that allows to detect possible errors during the execution of an integration solution. Our research results indicate that our proposal is easier to maintain than the current tools. To evaluate and demonstrate the viability of the contributions in this dissertation, we present five case studies to which we applied our proposal. The results in this dissertation have been transferred to a spin-off and have been published as three journal papers, seven conference papers, and three workshop papers.

