Fundamentals of Process Integration

by
Lachlan James Aldred
BBltEnv, MInfTech

A dissertation submitted for the degree of

IF49 Doctor of Philosophy

Principal Supervisor: Professor Arthur H.M. ter Hofstede
Associate Supervisors: Professor Marlon Dumas & Professor Wil M.P. van der Aalst

Faculty of Science and Technology
Queensland University of Technology
Brisbane, Australia

March 2011
Dedicated to my wife Thongkham Aldred for her patience, and to my daughter Monikha Aldred for her interest in all things scientific.
Keywords

Business process, integration, middleware, coupling, modelling, architecture, correlation, conversation modelling, interaction, patterns, messaging, languages, orchestrations.
Abstract

Technologies and languages for integrated processes are a relatively recent innovation. Over that period many divergent waves of innovation have transformed process integration. Like sockets and distributed objects, early workflow systems offered programming interfaces that connected the process modelling layer to any middleware. BPM systems emerged later, connecting the modelling world to middleware through components. While BPM systems increased ease of use (modelling convenience), long-standing and complex interactions involving many process instances remained difficult to model. Enterprise Service Buses (ESBs), followed, connecting process models to heterogeneous forms of middleware. ESBs, however, generally forced modellers to choose a particular underlying middleware and to stick to it, despite their ability to connect with many forms of middleware. Furthermore ESBs encourage process integrations to be modelled on their own, logically separate from the process model. This can lead to the inability to reason about long standing conversations at the process layer.

Technologies and languages for process integration generally lack formality. This has led to arbitrariness in the underlying language building blocks. Conceptual holes exist in a range of technologies and languages for process integration and this can lead to customer dissatisfaction and failure to bring integration projects to reach their potential.

Standards for process integration share similar fundamental flaws to languages and technologies. Standards are also in direct competition with other standards causing a lack of clarity. Thus the area of greatest risk in a BPM project remains process integration, despite major advancements in the technology base.

This research examines some fundamental aspects of communication middleware and how these fundamental building blocks of integration can be brought to the process modelling layer in a technology agnostic manner. This way process modelling can be conceptually complete without becoming stuck in a particular middleware technology.

Coloured Petri nets are used to define a formal semantics for the fun-
damental aspects of communication middleware. They provide the means
to define and model the dynamic aspects of various integration middleware.
Process integration patterns are used as a tool to codify common problems
to be solved. Object Role Modelling is a formal modelling technique that
was used to define the syntax of a proposed process integration language.

This thesis provides several contributions to the field of process integra-
tion. It proposes a framework defining the key notions of integration middle-
ware. This framework provides a conceptual foundation upon which a process
integration language could be built. The thesis defines an architecture that
allows various forms of middleware to be aggregated and reasoned about at
the process layer. This thesis provides a comprehensive set of process inte-
gration patterns. These constitute a benchmark for the kinds of problems
a process integration language must support. The thesis proposes a process
integration modelling language and a partial implementation that is able to
enact the language.

A process integration pilot project in a German hospital is briefly de-
scribed at the end of the thesis. The pilot is based on ideas in this thesis.
## Contents

1 Introduction
   1.1 Process Integration ........................................ 1
      1.1.1 Complexity ........................................ 3
      1.1.2 Trends in Process Integration ...................... 7
      1.1.3 Conceptualisation .................................. 8
   1.2 The Problem ............................................. 12
      1.2.1 Statement ........................................... 12
      1.2.2 Research Objective ................................. 13
      1.2.3 Solution Criteria .................................. 13
   1.3 Approach .................................................. 16
   1.4 Outline .................................................... 17

2 Literature Review ........................................... 23
   2.1 Foundations .............................................. 23
      2.1.1 Formalisms .......................................... 23
      2.1.2 Patterns ............................................. 25
   2.2 Distributed Systems Integration ....................... 27
      2.2.1 Trends ............................................... 27
      2.2.2 Standards .......................................... 32
      2.2.3 Technical Challenges ............................... 35
   2.3 Process Integration ...................................... 38
      2.3.1 Languages .......................................... 39
      2.3.2 Software ............................................ 48
   2.4 Conclusion .............................................. 50

3 The Dimensions of Loosely/Tightly-Coupled Architectures 53
   3.1 Introduction ............................................ 53
   3.2 Background ............................................. 55
   3.3 Distributed Systems Coupling ............................ 57
      3.3.1 The Thread-coupling Dimension .................... 57
      3.3.2 Time ................................................. 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.6 Filtering Patterns</td>
<td>197</td>
</tr>
<tr>
<td>6.3.7 Batch Messaging Patterns</td>
<td>198</td>
</tr>
<tr>
<td>6.3.8 Batch Filtering Patterns</td>
<td>198</td>
</tr>
<tr>
<td>6.3.9 Transformation Patterns</td>
<td>199</td>
</tr>
<tr>
<td>6.3.10 Discovery Patterns</td>
<td>200</td>
</tr>
<tr>
<td>6.4 Implementation</td>
<td>201</td>
</tr>
<tr>
<td>6.4.1 Architecture</td>
<td>202</td>
</tr>
<tr>
<td>6.4.2 e-Health Proposal</td>
<td>205</td>
</tr>
<tr>
<td>6.5 Conclusion</td>
<td>208</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 Epilogue</td>
<td>211</td>
</tr>
<tr>
<td>7.1 Chapter Overview</td>
<td>211</td>
</tr>
<tr>
<td>7.2 Solution Criteria</td>
<td>212</td>
</tr>
<tr>
<td>7.3 Further Work</td>
<td>215</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Three enterprises and a freelance agent are cooperating around an integrated acquisition process. ........................................... 2
1.2 A process integration stack. ......................................................... 18
1.3 Sample flight check in workflow. .................................................. 20

3.1 Blocking send. After initialising a send action, the transition “process” cannot fire until a thread is returned at the end of message transmission. .................................................. 58
3.2 Non-blocking send. The transition “process” can be interleaved with communication steps because a thread is not yielded to the embedded middleware. ................................................. 60
3.3 Blocking receive. A thread must be yielded to the embedded middleware until the message has arrived. ............................................. 62
3.4 Non-blocking receive. A thread need not be yielded to the middleware in order to receive. ...................................................... 62
3.5 Time coupling is characterised by transition-bounded systems. ... 63
3.6 Time decoupling is characterised by the presence of an intermediate endpoint. ................................................................. 64
3.7 Notation, space coupled. The sender directly addresses the receiver. 65
3.8 Extensions to MSCs representing two forms of space decoupling. 66
3.9 CPN presenting a semantics for the space dimension. The transition “begin x-port” has been annotated with a box labelled “Allocate”, which means that this transition decomposes to a sub-net (see Figure 3.10). ......................................................... 66
3.10 This CPN is the ‘Allocate’ sub-net refining transition “begin x-port” in Figure 3.9. ................................................................. 67
3.11 Modelling a thread-coupled, time-coupled, space-coupled interaction. 70
3.12 Modelling a thread-decoupled, time-decoupled, and space-decoupled interaction. ................................................................. 71
3.13 Notations for the coupling integration configurations – space-coupled communication. .................................................. 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14</td>
<td>73</td>
</tr>
<tr>
<td>3.15</td>
<td>74</td>
</tr>
<tr>
<td>3.16</td>
<td>77</td>
</tr>
<tr>
<td>3.17</td>
<td>78</td>
</tr>
<tr>
<td>3.18</td>
<td>79</td>
</tr>
<tr>
<td>3.19</td>
<td>80</td>
</tr>
<tr>
<td>3.20</td>
<td>81</td>
</tr>
<tr>
<td>3.21</td>
<td>82</td>
</tr>
<tr>
<td>3.22</td>
<td>90</td>
</tr>
<tr>
<td>4.1</td>
<td>106</td>
</tr>
<tr>
<td>4.2</td>
<td>108</td>
</tr>
<tr>
<td>4.3</td>
<td>109</td>
</tr>
<tr>
<td>4.4</td>
<td>111</td>
</tr>
<tr>
<td>5.1</td>
<td>127</td>
</tr>
<tr>
<td>5.2</td>
<td>128</td>
</tr>
<tr>
<td>5.3</td>
<td>128</td>
</tr>
<tr>
<td>5.4</td>
<td>130</td>
</tr>
<tr>
<td>5.5</td>
<td>130</td>
</tr>
<tr>
<td>5.6</td>
<td>131</td>
</tr>
<tr>
<td>5.7</td>
<td>131</td>
</tr>
<tr>
<td>5.8</td>
<td>132</td>
</tr>
<tr>
<td>5.9</td>
<td>134</td>
</tr>
<tr>
<td>5.10</td>
<td>135</td>
</tr>
<tr>
<td>5.11</td>
<td>135</td>
</tr>
<tr>
<td>5.12</td>
<td>136</td>
</tr>
<tr>
<td>5.13</td>
<td>137</td>
</tr>
<tr>
<td>5.14</td>
<td>138</td>
</tr>
</tbody>
</table>
6.21 Channel mobility .................................................. 182
6.22 Processes exploiting instance channels for correlation .......... 183
6.23 Conceptual syntax for channel grounding .......................... 185
6.24 Flight check in process (repeated from Figure 1.3) ............... 186
6.25 Purchase order selection assuming asynchronous quoting services . 187
6.26 Asynchronous purchase order selection using conversation-based filtering and timeout pre-event. ................................. 188
6.27 Asynchronous purchase order selection using instance channel messaging and timeout pre-event. ................................. 189
6.28 Coupling patterns ..................................................... 190
6.29 More coupling patterns ................................................ 191
6.30 Controlling message consumption in YAWL ....................... 191
6.31 Request response patterns .......................................... 192
6.32 In process instance A’1 task Request Quote sends its instance channel (denoted with an ‘i’) to process B task Receive Request. Process B discovers how to contact process A in task Discover channel and assigns it to variable. The quote message is sent back over process instance channel belonging to A’1. ................................. 193
6.33 Process Buyer and Seller both use a variable CorrToken which is inserted in incoming/outgoing messages in order to correlate messages to correct instances. To assist illustration the usually invisible data variable CorrToken is depicted graphically. The data perspective of YAWL has no graphical presence. The variable can be inserted into a message using XPath and/or XSLT. Task Receive Quote uses a filter to compare message data with process data. ................................. 194
6.34 Process Buyer and Seller correlate incoming and outgoing messages. The nodes on the diagram located between the two processes are scoped to the Process Context. Conversations, Channels and Properties are declared within the Process Container (see the ORM definitions in Figs. 6.14 and 6.20). ................................. 194
6.35 In this iteration of the quote scenario the customer’s quote reference is not enough to uniquely define the conversation – in particular for the seller. The customer ID needs to be combined with the customer's quote reference in order to define the conversation. This is depicted using a compound property at each stage of the conversation. ................................. 195
Nested within the conversation between the customer and retailer is the conversation between the retailer and each supplier. The inner conversation P.O. uses property values learned from both the outer and inner conversations by using a compound property. A more complete explanation of this example is provided in the section on modelling conversations (Section 6.1.6).

An external credit agency may send a message warning that the applicant has a risk of defaulting. This is not solicited (waited for). Receiving a warning causes the event-driven task Cancel Loan Appl to cancel the loan.

A courier company offers a service to any of its customers that allows them to change delivery priority or final delivery address. Change requests made after 10:00 AM are applied at the beginning of the following business day. Each day the next deadline and next morning dates are calculated. Requests to change the delivery address arrive on the channel. The receive task consumes messages as long as they conform to the business time constraints and routes received messages to the Update Delivery subprocess in order to change the delivery address.

Message multicasting.

Batch messaging.

Batch filtering.

More batch filtering.

Handle frequency spike: If more than one hundred alarms arrive within thirty minutes begin an alarm spike handling flow.

Transformation of process messages into wire-messages is automatically performed by JCoupling. An XML message in process A is forwarded by the send task onto the channel and through to the JCoupling Bridge. The JMS Adapter inside JCoupling injects the XML into a JMS object automatically.

Architecture of the JCoupling 2 Implementation. This diagram was authored by J. C. Kuhr as part of the collaboration over the JCoupling-2 implementation (2009).

Showing how it would be possible to integrate a sensor array in an ICU ward with an HL7 information system and a YAWL process. This diagram was authored by J. C. Kuhr as part of the collaboration over the JCoupling-2 implementation (2009).

A close up view of how the JCoupling architecture will support process integration in an ICU ward [93].
List of Tables

1.1 Previously published works. ................................. 19
2.1 Comparison of process integration solutions. ............... 39
3.1 Evaluation of selected middleware solutions and standards. 84
4.1 Relation corresponding to channel QuotesCh. ............... 111
4.2 Results of performance tests. .................................. 117
6.1 Subtype definitions for send tasks ORM diagram (in Figure 6.3). 161
6.2 Subtype definitions for receive tasks ORM diagram (in Figure 6.5). 167
6.3 Subtype definitions for the entities presented in Figure 6.14. . 176
6.4 Subtype definitions for the entities presented in Figure 6.18). . 180
6.5 Coupling patterns. ............................................. 190
6.6 Message consumption patterns. ................................. 191
6.7 Response patterns. ............................................. 192
6.8 Composed interaction patterns. .............................. 193
6.9 Event-driven patterns. ............................. 196
6.10 Filtering patterns. ............................................. 197
6.11 Batch patterns. .............................................. 198
6.12 Batch filtering patterns. ..................................... 199
6.13 Transformation patterns. ..................................... 200
6.14 Discovery patterns. ........................................... 201
6.15 Implementation coverage (+ = supports, +/- = partial support, = not yet supported). 205
6.16 Implementation coverage (+ = supports, +/- = partial support, = not yet supported). 206
Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed: ____________________________

Date: ____________________________
I would like to gratefully acknowledge the principal supervision of Dr. Arthur ter Hofstede. The tuition in scientific techniques and the endless effort in perfecting the ideas are all appreciated. This thesis would not have been possible without the transformative ideas from my associate supervisors Dr. Wil van der Aalst and Dr. Marlon Dumas. The technical discussions, insightful feedback and contributions are all deeply appreciated.

I owe my deepest gratitude to Dr. Jan-Christian Kuhr, Thomas Becker, Marcel Paleit, Tony Möller, Jan Pretzel and Lukas Vedral based at Gecko, in Rostock, Germany. Without their software engineering efforts YAWiL would just be a notation.

I am grateful to Howard Triesman and all my friends and colleagues at Avoka Technologies. The technical discussions and battlefield experience gained in Australia and the United States helped ground the ideas in this thesis.

This PhD was partly funded by an Australian Research Council Discovery Grant “Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages”.

Acknowledgements
Chapter 1

Introduction

The challenges faced by stakeholders of Business Process Management (BPM) systems are numerous. There is a diverse array of technical, cultural, and conceptual issues to deal with. Potential points of resistance include the political impact of changing the way people do their jobs – brought about by office automation. Also the simple immaturity of the solutions available make selection of the right solution confusing for customers. There are conceptual limitations. One example is that business rules are hard to define on how to react to a complex set of business events. An example of a technical limitation is the integration of processes over heterogeneous networks and software.

Despite these pitfalls investment in BPM technology by adopters appears to be increasing at an ever greater rate. This is probably because of three primary driving forces: (1) that companies feel the increasing pressure of a globalizing market; (2) customers have become more empowered and have relentless demands for quality, service, and price; and (3) the progress of Information Technology has enabled the possibility of managing business processes better and with increased efficiency [123]. Beyond the opportunities for managing processes, Information Technology has also facilitated the process of creating processes, through abstract, graphical languages and sophisticated modelling tools. The driving goal is to create these processes codelessly. This is only made possible through effective conceptualisation of the problem domain - business processes.

1.1 Process Integration

The more a process aims to achieve, the less likely it can operate as an island. In order to survive, businesses need to cooperate with their customers, and
with other business processes. Naturally it follows that sophisticated business processes need to cooperate with other businesses; i.e. customers, workers in partner-organisations, and remote processes. Processes, indeed, need to operate across organisational boundaries. These boundaries may exist between departments, organisations, and/or across geophysical space. Each cooperating process, running in its respective department/organisation, may form a mere part of a greater process that spans many environments when viewed holistically. *Process Integration* enables processes to effectively perform their intended purpose, irrespective of these boundaries. For instance Figure 1.1 shows that several processes may be interacting over a longstanding activity such as an acquisition by a buyer. The interactions span three enterprises and a freelance Buyer’s Agent. Note that Figure 1.1 shows the Buyer’s Agent and New Retailer as individuals working outside of a process enactment environment. Nevertheless they are able to participate *seamlessly* in the process using a laptop (perhaps running email) and a hand-held device.

![Figure 1.1](image_url)

Figure 1.1: Three enterprises and a freelance agent are cooperating around an integrated acquisition process.

Stand-alone business processes are model-driven, not coded, likewise there
is a driving desire to model and enact integrated business processes in a similar fashion. The modelling environments in BPM software have been allowing users to specify processes codelessly for a number of years. These BPM offerings usually provide an abstract, graphical environment for model-driven integrated processes. However, in the field of *model-driven process integration* many issues still remain. A major contributor to these remaining issues is the innate complexity of integrating business processes.

**1.1.1 Complexity**

Integrating business processes produces a raft of issues, at many levels. This section highlights some of the issues (for instance one needs to resolve the heterogeneity of networks, messaging technologies, and of BPM software).

**Availability and Latency**

Unlike modern, wired networks, wireless-mobile networks experience sporadic availability and higher latency. As more and more business is conducted with mobile staff/partners wireless-mobile networks are becoming far more frequently used for conducting business. Like wireless networks, Virtual Private Networks (VPNs) also experience latency despite their popularity. VPNs are mentioned because they do get used in integration scenarios and they have such a massive impact on network speed (up to an order magnitude slower). Indeed integrated processes need to be able to operate over such networks, despite their problems.

**Heterogeneity**

Middleware technologies have been around for decades. As successive waves of middleware technologies went to market they varied hugely in terms of what they did and how they did it. For instance, the Distributed Object technologies such as Microsoft’s DCOM [61] and CORBA [109, 83] were programming language-oriented, *highly responsive* and made the remote object appear local. Conversely the next generation of middleware technologies included Message-oriented middleware (MOM). MOM such as Microsoft’s MSMQ [38] and IBM’s MQ Series [79] were programming language agnostic, asynchronous and made any remote object invisible.

Enterprise Service Bus (ESB) software succeeds MOM, and generally enables application integration through short-lived orchestrations. ESBs typically support centralised message management, auditing, and conceptual
modelling in a drag-and-drop environment. However ESB tools address middleware technology heterogeneity by using heterogenous plug-ins (one for each middleware technology), and consequently appear somewhat ad-hoc – from a conceptual perspective.

Included in the array of technologies being used for integration are ubiquitous, cheap, highly available messaging technologies such as FTP and Email. These are being used by system administrators to integrate applications without any expensive software and thus are invisible to the business [147].

An interesting edge-case of heterogeneity occurs when the remote process, or parts of the local process, are not automated. The exclusive use of BPM technology in all remote sites is simply not realistic. For instance government to citizen process integration is only viable using email, Fax etc. When there is no software operating at the remote site it is additionally challenging to understand the context of an incoming message. All of the calculations for determining the corresponding process instance to route the message to, must be done by ‘gleaning’ context from the document/text etc.

**Uncertainty**

Cooperating processes will at times need to know what the other is doing or whether a message sent out has been received and understood by the partner process. Integrated processes operate independently and this causes uncertainty of the exact state of the remote process. In mission critical situations, this is a fundamental issue to be solved.

The ‘Two Generals Paradox’ describes a scenario where two generals and their armies are separated by a valley containing enemy soldiers. They plan to attack a city however both need to be certain of each other’s intentions and timing. Only a concerted, synchronised attack would result in victory. An attack by one general and not the other would cause the attack to be repelled by the enemy. The problem is that coordination of the attack, and its timing, is only possible by sending messengers through the enemy occupied valley. There is no way of being certain that a messenger reached the allied general. Upon examination there is no sequence of messages that can allay one general’s uncertainties about the other general’s intentions which was proven by Gray [67].

Another factor creating uncertainty is the possibility of ‘traitors’ (in a non military situation this could be construed as errors in remote process models) which cause messages to contain misleading data. This is described as the ‘Byzantine Generals Problem’ by Lamport et. al. [94].

Indeed, there is uncertainty about the actions and intentions of remote
processes. To minimize this uncertainty (but not remove it) messaging technologies employ feedback mechanisms such as remote exceptions, event-handlers, and request-response. Naturally, a process integration design needs to show relevant aspects of these communication techniques at the modelling (conceptual) level.

**Diversity of Coupling Styles**

Integrated processes, depending on their purpose, require different coupling styles. For example, when a telecommunications company experiences a cut to one of its underground optical fibres the alerts being generated will require immediate handling. Telephone traffic will need to be sent to alternative routes immediately. Needed is a process integration solution that can monitor events (or sets of events) and take action when required, immediately checking the success/failure of the action taken.

Conversely, the transaction processing systems running in banks rely on loose couplings with the systems in other banks. For instance, interbank transactions are traditionally reconciled overnight.

The differences in coupling style are the outcome of divergent business needs: immediacy for the telecommunications company versus reliability for the bank. *Process integration software needs to be flexible enough to accommodate these differences.*

**Responding to an Ever Changing Business**

The context of doing business is constantly changing. It follows that these constantly occurring changes need to be addressed by active processes, but the process cannot wait for them because the events don’t always occur in every case. If these changes are able to be converted into events a workflow system can catch and handle them.

Processes need to be able to connect with event sources. Events may start process instances or influence running instances. A business may be eager to receive a change notification. In such a case it would stop and wait for the change event. Alternatively a business process may be able to progress without an event; in which case the process should be able to not wait for the event but do something if the event is generated. Examples of events that a process may be interested in include:

1. An incoming purchase order.
2. A customer relationship management system announcing that an insurance contract approaches its renewal date.
3. A concentrated set of equipment failures of an airline’s fleet.

4. A telephone company detects a SIM card making a phone call from Sydney. Five minutes later an apparently same SIM card, makes a call from London (SIM fraud).

5. A bank observing many credit-card purchases, each using a different credit-card, to the same shipment address.

6. A significant acquisition of a publicly traded stock, by a small set of buyers, followed, the next day, by a share price changing announcement by the board of directors.

Example 3 may signal that some careful management and monitoring is required of maintenance procedures for an airline’s fleet. Examples 4, 5 and 6 may indicate fraud. Note that examples 3, 4, 5, and 6 all involve a set of events occurring around the same time and having some relationship to one another. Encoding the business rules for capturing and acting on these sorts of events is extremely difficult, even when using state-of-the-art process integration software.

Large Numbers of Messages

When large sets of messages need to be handled it would be conceptually fitting to receive and handle them en-masse. Sending or receiving hundreds, or perhaps thousands of related messages may be necessary at times, for example an alarm handling process in a telecommunications enterprise may receive hundreds of messages at once. Indeed, process integrations need to support handling batches of messages.

Conversation Modelling

Messages directed at one of many instances of the same process generally flow on shared enterprise messaging resources. Having a discrete resource (such as a URL) dedicated to each process instance is unusual, even if it is desirable [156]. Business processes, being massively parallel, require incoming messages to be correlated with, and routed to, the appropriate process instance.

Cookies are a well known solution to this problem in the field of Web design. Cookies store enough context about a sequence of interactions that the Web server can discern incoming requests concerning different customers.
Cookies are non-trivial to develop. A Web-site developer using cookies requires extensive technical knowledge, extensive coded effort, and significant testing. In business none of these are in great supply.

Technically, correlation is one of the most significant challenges for integrated processes. There are four sets of conceptual problems related to correlation:

1. How does one compose outgoing messages in order to make sure that these messages are correlatable inside the remote process?

2. What needs to be done in order to interpret incoming messages the right way, making them route-able to the right process instance?

3. How are the requirements for correlation between partners modelled and communicated, such that both can effectively comply?

4. How do we deal with change management, between different organisations, when one partner wants to change the structure of messages being sent?

1.1.2 Trends in Process Integration

The first workflows were able to coordinate the actions of people in an organisation. Workflow systems matured to coordinate the actions of both people and applications. Early workflow systems provided interfaces allowing interactions with invoked applications and interfaces for inter-operation with other workflow enactment services. This idea is observable in the Workflow Reference Model (WRM) authored by the Workflow Management Coalition [149] in 1995. The workflow reference model defined five standard interfaces for a workflow system.$^{1}$ Two of the five interfaces (3 and 4) focused on aspects of process integration. Interface 3 addressed the need for processes to invoke remote applications, while Interface 4 enabled a process to interact with remote workflow systems. This interface-based process integration architecture provided process integration developers with APIs to design/develop/deploy software components that interconnected a process with remote applications/processes.

Later, BPM systems became available that typically supported *turnkey* integration. Supplied with the BPM product were *commoditized*, re-useable adapters, that could be purchased off-the-shelf and could bridge into other systems. An example of an off the shelf adapter would be one that bridges into SAP\(^2\) or to Siebel \(^3\). This small step forward in conceptualisation represented major progress for business customers. This is because they could perform basic bridging functions without having to write any code.

BPEL4WS 1.0 \([47]\) was released in 2002 and it employed a Service-Oriented Architecture (SOA) approach to process integration. *Invoked Applications* (Interface 3 - WRM) became Web-service invocations. And the process itself was able to be invoked as a Web-service (*Other Workflow Enactment Services* Interface 4 - WRM). Thus the SOA approach defined in BPEL was markedly different to the WRM. The process itself could be exposed as a service to other remote processes and applications through a consistent architecture.

At the conceptual level, Correlation-Sets in BPEL addressed conversations in a conceptual manner. Correlation-Sets allow process integration modellers to extract correlation values out of fields that are buried deep inside an incoming message.\(^4\) This is a significant departure from using generated correlation IDs in message headers, and passing them backward and forward, as was done in traditional EAI \([77]\). Thus, in BPEL4WS, and its successor WS-BPEL \([18]\), there was significant progress towards a more abstract, model-driven approach to process integration. However, BPEL’s syntax is XML-based making it difficult to understand. BPEL does not support correlating messages from different sources or batch messaging directly. Furthermore, BPEL possibly does not hide less relevant technical detail about access and middleware technology, due to its dependence on the Web Service Description Language (WSDL) \([108]\).

### 1.1.3 Conceptualisation

There are already many languages and tools for modelling and enacting process integrations. For these offerings to be useful and helpful it is essential that they align with process concepts and integration concepts. It would indeed be a major problem if these offerings had conceptual holes in them.


\(^4\)For example a Purchase Order ID could be extracted out of an XML field inside a message body.
The process of conceptualisation is the process of abstracting a set of ideas, about a particular problem domain, based on specific instances [81]. Thus a conceptualisation of process integration would capture its essential entities and their static and dynamic relationships with one another. Conceptualisation reduces the amount of effort involved in expressing and understanding instances of a problem domain. Abstracting the concepts of integration enables technology independence. By capturing the essence of integration in a conceptualisation there exists the potential to use any integration technology i.e. no technologies are unnecessarily ruled out. Also abstraction, done well, facilitates deployment and maintenance of process integrations. Therefore, the conceptualisation of process integration is a necessary precursor to modelling.

Clearly there is a strong preference for graphical modelling tools in most BPM products. The same preference for graphical modelling environments is appearing in business process integration tools. Indeed graphical modelling tools for process integration enable solutions to be created codelessly, by employing conceptual representations of messaging resources, process actions, messages, and remote systems.

The conceptualisation of a problem domain, unfortunately, can be performed poorly. One reason for this is that the problem domain is complex. Process integration is indeed complex. It appears to be the case that process integration products have thus far conceptualised process integration in an ad-hoc manner. Likewise, distributed systems integration tends to be characterized by wave after wave of new ideas that carefully try to avoid repeating the ‘mistakes’ of the last wave. This tendency has been compared (humorously) to a drunk driver constantly swerving to avoid hitting kerbs on opposite sides of a road [75]. Perhaps failing to find the right abstractions for distributed systems integration may be the cause of general integration problems.

**Principals of Conceptual Modelling**

The conceptual modelling community understands the difficulty, and the importance of finding the right abstractions to a given problem domain. A guiding principle for this process is expressed in the ISO Technical Report 9007 [81]. This guiding principle states that apt (suitable) abstractions must convey essential information about the domain as well as being convenient i.e. permitting easy manipulation to meet specific needs of all scenarios. This notion of convenience (or suitability [74, 91]) of a conceptual model means that creators of process integration models prefer ideas that have strong alignment with the problem domain, and can be manipulated easily enough.
to address the more difficult scenarios.

ISO Technical Report 9007 also discusses two general principles:

**The 100 Percent Principle** which states that all *relevant* static and dynamic aspects of the domain (i.e. its nature) should be described in the conceptual schema. Relevant aspects that need to be modelled outside the sphere of control compromise the *holistic* model.

**The Conceptualisation Principle** states that a conceptual schema should only include relevant aspects. This is to the exclusion of aspects related to lower level technical details. In the case of process integrations this would include details concerning physical access to enterprise integration resources, message formats, and communication protocols.

The conceptual nature of an abstraction enables hiding the details of what is being abstracted from. That is, in essence, the compelling argument for using them. Perhaps there is a balance to be found between the hiding of irrelevant details (the Conceptualisation Principle) with the complementary force of keeping the conceptualisation true to its fundamental semantics (the 100 Percent Principle).

**Software Abstractions**

The software development community are also aware of the difficulty and importance of finding the right abstractions through careful selection of the right API for a given task. Their frustrations at inapt APIs result in an endless procession of new APIs replacing preceding designs.

An interesting commentary on the importance of finding suitable abstractions is expressed by Spolsky. Spolsky is known in the developer community for writing the *Law of Leaky Abstractions*: “all non-trivial abstractions, to some degree, are leaky” [1]. The rationale behind his assertion is that there is no perfect implementation of an abstraction. Quality abstractions, should speed up development times, by reducing the amount of code that needs to be written. But quality abstractions should also reduce the amount of learning needed to be done, as they reveal only necessary detail, hiding the unnecessary. Spolsky argues that leaky abstractions may speed up the process of building the solution immensely, but when something goes wrong and the solution needs to be debugged, it can take weeks of learning in order to troubleshoot and fix a single issue.
Process Integration Abstractions

When a business process delegates integration to a component, it abstracts from the integration technology. Nevertheless there are a host of potential problems (e.g. a network exception, a lost message, a duplicate message, a receive timeout). Techniques in distributed systems integration overcome these issues (such as fault handlers, guaranteed delivery, message IDs, etc. [77]) and it is essential that these can be modelled in a process integration. Process integration components should follow the 100 Percent Principle, and therefore these techniques must be exposed to the process modeller.

Process integration is an extremely technical problem, however designs can be overloaded with less relevant details. For instance whether a message needs to be sent over JMS [70] versus MSMQ (two fairly equivalent techniques) is just a technological concern. Other aspects that are not relevant conceptually include the middleware server address and port number. These aspects should be captured as configurations to a specific process integration runtime environment. Likewise construction of a suitable message object (e.g. the SOAP header and SOAP Envelope) are not conceptually relevant. Therefore it is essential for purposes of brevity and process portability that references to: specific messaging technologies, message object creation/structure, and how/where to access middleware instances do not need to be added to the process integration model. The Conceptualisation Principle suggests that such low-level concerns are not required in the process model.

Roadblocks to Conceptualisation

In addition to the sheer complexity of process integration, problems and weaknesses of existing products create additional challenges.

Weak meta-models undermine the foundation of any model. A strong conceptual model needs to be expressive, precise, conceptual, suitable and understandable. These criteria are explained in Section 1.2.3. The right abstractions help shape the thinking of architects and developers towards the optimal solution, but in order to achieve this the meta-model needs to be convenient to use. Unfortunately meta-models for distributed systems integration continue to frustrate users [11, 15, 27]. Naturally weaknesses in the meta-models for distributed systems integration are inherited by process integration meta-models.

Immature modelling software is prone to burden a process integration model with less relevant details and possibly handle the more relevant aspects of process integration poorly (violating the Conceptualisation Principle and
the 100 Percent Principle). At this time the Enterprise Service Bus (ESB) phenomenon categorises the current generation of process integration tools available. Many of the ESB products expose less relevant details (e.g. a particular middleware technology, message object structure, server address, etc.) to the process layer.

ESBs, as mentioned already, generally resolve heterogeneity of middleware/applications using heterogenous adaptors. Consequently each process step is tightly-coupled to its respective adaptor technology, making the model less conceptually-aligned with the problem domain and more aligned with technology. This violates the Guiding Principle of conceptual modelling and results in problems with suitability (see Section 1.1.3). Also, while many ESBs allow conversations to be modelled, ESB models are not generally regarded as business process models. The two are generally executed and run in different servers. Conversations between role players are abstract designs and thus should be modelled in an abstract context (not necessarily in the ESB).

Forced compartmentalisation occurs where concepts that belong together are being arbitrarily separated into unnecessary layers. It is well understood that too few layers causes problems, however too many brings about problems as well. This is particularly the case if the boundaries between layers are poorly drawn. Indeed, “all problems in computer science can be solved by another level of indirection” [135], but as noted by Wheeler “that creates another problem”.

For example the use of a dedicated ESB product deployed separately to a dedicated Workflow product forces compartmentalisation between the human-oriented and the integration-oriented aspects of a business process. Hiding relevant information in the modelling environment is a violation of the 100 Percent Principle.

1.2 The Problem

1.2.1 Statement

Process integration is complex. There are issues concerning latency, unreliability, heterogeneity, uncertainty, heterogenous coupling techniques, changing businesses, large numbers of messages and conversation modelling.

Despite the strong desire for model-driven approaches to process integration, tools and languages are ad-hoc, they lack maturity, are constantly changing, are highly heterogenous (depending on which vendor) and are conceptually weak. Tools and languages for process integration frequently over-
look or ignore all of the complexities process integration thus many aspects are simply absent in their conceptual models.

Many languages, for instance address the highly abstract issues but overlook the less abstract structure of an interaction (violating the 100 Percent Principle). Many tools, on the other hand, address the less abstract issues but bind too heavily into a particular technology. This can occur to such a degree that the these bindings become rigid (violating the Conceptualisation Principle).

There is a lack of balance and correct emphasis in process integration. This imbalance is observable by the tendency for languages and tools to encourage a separation of workflow models from integration models – forcing a fractured paradigm on the unsuspecting user community.

1.2.2 Research Objective

This research aims to analyse issues concerning the conceptualisation of process integration. It is intended that a thorough, conceptual analysis of process integration will lead to a higher quality process integration meta-model than that which exists in currently available model-driven process integration products. The intended outcome of this research is a highly conceptual, expressive language for creating effective process integration models.

1.2.3 Solution Criteria

The fundamental precursor to the conceptualisation of process integration is finding the right meta-model. Therefore it is appropriate to borrow criteria from the conceptual modelling community for assessing meta-models. The following categories of criteria for process integration assessment (being Expressive, Precise, Conceptual, Suitable and Understandable) are adapted from ter Hofstede’s thesis [74].

Expressive

Essentially a meta-model for process integration should capture close to 100 percent of the relevant dynamic and static aspects of process integration.

Specifically, these include:

- Can a business process be modelled to integrate with manual technologies such as Facsimile (and perhaps manual remote processes)?

- Is the way the interaction occurs over the underlying middleware visible at all in the process model?
• Is the success/failure of an interaction made clear or made opaque to the process layer?

• To what degree can a process integration model specify synchronous/asynchronous and other well-known patterns of interaction, such as publish-subscribe, request-response, fire and forget?

• Can certain combinations of messages, from totally different sources be selected together if they are related to one another in real world terms?

• Is it possible to model that a process only reacts to a certain combinations of events?

• Is it possible to model the difference between a mission-critical integration and a process integration that does not need to be available all the time?

• It is possible to model process integrations using inexpensive, readily available protocols, e.g. email and FTP?

• Can conversations be modelled, linking together many messages from many sources; regardless of heterogenous message formats, and irrespective of middleware?

• Does a synchronous or asynchronous interaction look different at the process layer?

• Is it possible at the process modelling layer to determine which capabilities are available, depending on which middleware is being used?

**Precise**

Ultimately, these models must be executable on hardware. Therefore what is modelled must be precise. Ideally, the creator’s understanding of the meta-model should be identical to a process integration modeller’s. Such an outcome is really only possible if the meta-model is precise, unambiguous, and without hidden surprises. Otherwise there will be greater technical risk, greater development effort, and transformation challenges at end-of-life (when migrating models to a new environment).

A formal definition of the meta-model (firstly defining static structure, but preferably defining structure and dynamic semantics) is an excellent means for ensuring precision.
Conceptual

Conceptualisation is an essential criterion for measuring how the proposal manages the sheer number of details in a process integration model - by selectively exposing the relevant and hiding the less relevant.

Specifically, a strong proposal would provide positive responses to the following questions:

- Is it possible to express a model of process integration that can potentially plug-into, and out-of, alternative forms of middleware?

- Is it possible to specify a well-known interaction pattern (request-response, publish-subscribe etc.) without specifying a middleware technology?

- Is it possible to hide details such as message object format, access locations of middleware, and middleware logins from the process integration model?

- Is the conceptualisation flexible enough to enable message selection/conversations without making demands on the message sender in terms of format, and without making any limitation to the type of middleware being used?

A solution would need to potentially combine a wide array of integration technologies and protocols. However this should not be in conflict with the innate need to be abstract. Consequently, burdensome details such as details of middleware protocols, middleware server addresses, usernames/passwords, over-the-wire message formats, and configurations should not invade the process integration model. These details, while being absolutely essential to the integration solution clearly belong at a lower layer of abstraction. In doing so it would be possible to substitute one middleware solution with another without forcing too many unwanted changes to the process model.

Suitable

Suitability is important as a solution criterion because it encapsulates the notion that the meta-model has strong alignment with the problem domain, and that it allows a domain to be modelled in a direct manner [74, 91].

Questions such as the following could assess the alignment of a process integration language with the problem domain:
To what degree can a process model effectively rule-in certain types of middleware and rule-out others; based on the match between the modelled intent and the respective middleware capabilities?

- How much modelling effort is required to process large sets of messages?
- How much modelling effort is required to select messages based on their contents?
- How much effort is required for messages to be selected (or filtered), regardless of middleware technology and message format?
- How much modelling effort is required to model a conversation from many interactions?
- How much effort is required to specify what determines a certain set of messages forming a conversation?

**Understandable**

Despite the innate complexity of process integrations, they will inevitably need to be shared among stakeholders in the business. Therefore, it is highly desirable that models of process integration can be explained and understood by project outsiders as quickly as possible.

- Are models able to be expressed sufficiently concisely that they can be shared with others?
- To what extent does the process integration language have a graphical syntax?
- Is it possible to create a holistic view over a process integration?

### 1.3 Approach

This course of research begins with a detailed analysis of existing forms of middleware used in enterprises. There appear to be many missing aspects to the conceptualisation of basic integration. This motivates a bottom-up approach to its analysis.

Coloured Petri-nets (CPNs) [87] are a suitable technique for analysis of basic integration. The strengths of CPNs include that they are extremely effective at conceptualising highly parallel systems, and that well known techniques and tools can be employed to test those models leading to conclusions
and assertions about the problem domain. These strengths of CPNs are highly aligned to the conceptual and technical problems and issues of process integration. Other formal techniques for integration analysis, such as π-calculus [102] (and other Process Algebras), are worthy alternatives. However, while π-calculus is designed for distributed, parallel systems; Process Algebras do not have a graphical representation, and are not able to capture the notion of state and parallelism as succinctly as CPNs.

In order to underpin a conceptualisation of process integration it is necessary to conceptualise the notion of coupling. The notion of coupling, (1) is generally not easily expressed previously in a formal manner, and (2) exists at the heart of every integration model. The analysis of coupling also leads to coupling/decoupling patterns (akin to the workflow patterns [7]) that can be used in incorporating coupling into a process integration language.

Proving the correctness of an architecture/meta-model is extremely difficult. Consequently a prototyping approach is used to demonstrate that a unified meta-model over coupling is feasible and can perform any conceivable form of coupling/decoupling.

Correlation, being a major challenge for modelling integrated processes, will lead to a CPN analysis of filtering messages. Message filtering, is widely supported in middleware technologies, and is under-utilized in message to process instance correlation. Once again a CPN analysis and prototype assists in shaping ideas and the proposal.

Analysis patterns concerning integration [77] and process integration [26], and correlation [25] were conducted by others prior to, or in parallel with this thesis. Patterns are widely accepted as a selection instrument for complex business systems [7], and in forming the requirements around a workflow perspective. This course of research proposes a set of process integration patterns, building on the coupling patterns, correlation, and core aspects of process integration. These patterns could also be useful in guiding the design and ascertaining the strength of a language for process integration modelling.

1.4 Outline

Figure 1.2 presents an overview of the respective layers of a process integration enactment stack. The Business Process layer is concerned with the modelling and execution of process integrations. The Message-Matching layer is responsible for bridging workflow with messaging. It also conveys messages to/from the right process instance. The Middleware Aggregation layer presents a consistent and meaningful view of heterogenous middleware to the upper two layers.
This thesis contains original work, however the majority of the content is drawn from previously published works. These are presented in Table 1.1.

Chapter 2 reviews literature and influences in the fields of application integration, middleware, and trends around BPM, SOAs, and in Enterprise Service Buses. It also discusses the impact of these fields on the conceptualisation of process integration.

Chapter 3 analyses a conceptual integration layer that allows different forms of middleware to be aggregated without neutering their respective strengths and weaknesses in terms of integration coupling (see Figure 1.2 Middleware Aggregation layer). Coupling exists as a core concern for any integration model. For instance, the decision to loosely or tightly couple two systems is frequently understood to correlate with a ‘synchronous’ or ‘asynchronous’ interaction model. However, as this chapter demonstrates, there are multiple dimensions to loose/tight coupling and synchronisation is just one. This chapter presents a framework for understanding integration coupling. That framework can be used to assess middleware capabilities, and
<table>
<thead>
<tr>
<th>No.</th>
<th>Citation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[11]</td>
<td>Formalises the notion of tightly coupled and loosely coupled architectures. This paper was nominated with honourable mention for the 2005 IFIP TC2 Manfred Paul Award.</td>
</tr>
<tr>
<td>2</td>
<td>[14]</td>
<td>Extends a previous paper (1). It formalises bi-directional interactions and presents a prototype flexible coupling service.</td>
</tr>
<tr>
<td>3</td>
<td>[15]</td>
<td>Extends two previous works (1) &amp; (2). It refines previous works and incorporates the notion of interaction timeouts into the formal models.</td>
</tr>
<tr>
<td>4</td>
<td>[13]</td>
<td>The former work describes an approach and an architecture for defining and executing batch message filtering. It also presents an implementation supporting the ideas therein.</td>
</tr>
<tr>
<td>5</td>
<td>[16]</td>
<td>Is an extended version of (4).</td>
</tr>
<tr>
<td>6</td>
<td>[10]</td>
<td>This work defines a set of patterns for process integration.</td>
</tr>
<tr>
<td>7</td>
<td>[105]</td>
<td>Defines a conceptualisation for complex request-response interactions that involve more than two endpoints.</td>
</tr>
<tr>
<td>8</td>
<td>[93]</td>
<td>Describes an approach for using message aware workflow technology to enable a vision for a smart hospital. It proposes the use of proximity aware devices to trigger workflows and automatically update hospital information systems.</td>
</tr>
</tbody>
</table>

Table 1.1: Previously published works.

thus has the potential to be used as a middleware selection instrument. The chapter presents a prototype implementation of this conceptual framework. This material draws from several of our previous publications [11, 15, 14].

Having examined the core concern of any integration model we examine a layer closer to the business process. This layer addresses the problem of matching messages to the right process instance. Chapter 4 proposes a generalisation to the BPEL correlation set, enabling correlation, message filtering, and composite messaging (see Figure 1.2 Message-Matching layer). Correlation, message filtering, and composite messaging, are all fundamental to process integration and all involve the selection of certain message/s over others – based on rules. Hence, this chapter proposes a message selection service underpinning these process integration techniques in the form of a shared message match-making service. The chapter also describes an implementation. The message match-making service, as designed, provides significant conceptual and performance advantages over a naive approach to message filtering. The content of this chapter is drawn from several of our
previous publications \[13, 16\]

Chapter 5 proposes a set of process integration patterns. These patterns generally specify what the static and dynamic capabilities of a conceptualisation to process integration are. They effectively set a benchmark for what a process integration language should be able to express easily and concisely. The content of this chapter draws from two of our previous publications \[10, 105\].

![Sample flight check in workflow.](image)

Figure 1.3: Sample flight check in workflow.

Chapter 6 describes the elements of a language for process integration (see Figure 1.2 Business Process, Message-Matching and Middleware Aggregation layers). This language builds on findings from previous chapters enabling workflow models to incorporate process integration.

Figure 1.3 presents a sample flight check in process modelled using this process integration language. The pipe symbols are called *channels* and are used to convey messages. A dotted line is used to connect *channels* to *communication tasks*. Communication tasks convey data in and out of the workflow in the form of messages.

Using this language process integration models can be designed using loosely-coupled and tightly-coupled approaches. The language can also easily express compound conversations, message correlation, composite messaging, and batch handling. The language addresses problems of technology heterogeneity in a conceptually clean manner. The proposed language is a
set of extensions to Yet Another Workflow Language (YAWL) for the purpose of conceptualising process integration. Chapter 6 also presents how the language covers the process integration patterns (mentioned in Chapter 5). Finally, Chapter 6 presents *an implementation that is able to execute core aspects of this process integration language*. The discussion around this partial implementation draws from work previously published elsewhere [93]. This publication led to a patent application [92] for a hospital information systems integration in an intensive care unit.

Chapter 7 concludes the work.
Chapter 2

Literature Review

We seek a conceptualization of process integration that is powerful and yet convenient to use. There are a wide range of fields of endeavour focussed on bettering process integration. These range from efforts to understand the problem (e.g. patterns), through to languages expressing what we already know about the problem (e.g. formalisms), through to technologies resolving interoperability issues (e.g. architectures), and standardization efforts. Many of these are briefly described in this chapter.

2.1 Foundations

Formal languages for process integration attempt to allow us to model, describe and analyse specific aspects of a set of integrated processes. Patterns are an effort to describe and understand a given problem domain. There are several sets of patterns around process integration and this section contains a brief discussion of them.

2.1.1 Formalisms

Petri nets, Process Algebras and Message Sequence Charts are examples of formal techniques for modelling process integrations. Some formalisms offer analysis techniques over integration models. Each is interesting and relevant.

Petri nets

Petri nets [86, 54, 106] are a powerful, formal language for expressing parallel computation. They have a graphical representation, and although the core constructs of the language are minimal and simple they are capable of simulating massively complex, unbounded, parallel systems. The transitions
of Petri nets are analogous to tasks in business processes. This is one of the reasons that Petri nets have been proposed as being extremely suitable for modelling workflows [6, 5, 99, 2, 91]. Petri nets, have an active research community enhancing the already extensive knowledge base. Properties of models (e.g. deadlocks, livelocks, unbounded etc.) can be determined using Petri net analysis techniques before process execution. Coloured Petri nets (CPNs) are an extension of Petri-nets that enable the modelling of data, hierarchy and time. CPNs inherit Petri net analysis techniques.

Despite the analogous relationship between a Petri net transition and a workflow task there is no similarly intuitive mapping between a Petri net construct and an integration concept. This fact is not an assertion of Petri nets being unsuitable to model process integration, in fact the reality is that Petri nets are extremely suitable, but it is clear that the mapping is not as trivial as mapping workflow tasks onto transitions. The modelling of integration requires giving consideration to the interfaces that separate autonomous systems. These complex boundaries can be represented using Petri nets using either a transition-bounded (two autonomous systems sharing common transitions) or a place-bounded (sharing common places) approach. Not many Petri net tools support both transition bounded, and place bounded integrated nets. For instance CPN Tools [86, 45] supports place-bounded nets.\(^1\) Renew [124] supports transition-bounded intercommunicating nets only. Despite these challenges Petri nets have been proposed to provide an approach for modelling messaging interactions between multiple processes at the global level [8], but that approach chooses to represent the integration as a place and two arcs, thus trivialising the problem.

**Process Algebras**

The Process Algebras, for example the Algebra of Communicating Processes (ACP) [23], the Calculus of Communicating Systems (CCS) [101], and Communicating Sequential Processes (CSP) [73] are formal frameworks specifically created to model concurrent communicating processes. Each of these formalisms, while markedly different in terms of syntax, support analysis techniques including bisimulation and deadlock detection.

Milner’s \(\pi\)-calculus [102] has been cited as a formal foundation for business processes [117, 31]. \(\pi\)-calculus is noted for its ability to model mobility of channels between processes. \(\pi\)-calculus influenced a proposal for a logic-based language describing interactions across distributed entities [28]. Nevertheless

\(^1\)CPN Tools place-bounded nets use places as connection points to a transition that can be substituted with a sub-net. The transition and sub-net share the same I/O places.
π-calculus does have its own challenges with respect to representing some processes that are easily represented in Petri nets [3].

π-calculus is a suitable and purpose-built formal foundation to modelling process integration. It does not provide suitable modelling concepts for: complex message datasets, synchronous/asynchronous transfers, request-response, message filtering, correlation, etc, although these could be added.

Summary

We have chosen Coloured Petri net-based modelling techniques to define some formal foundations to integration modelling in Chapter 3. The graph-based syntax of CPNs, and their expressive power, make them a suitable candidate for conceptualising fundamentals of integration.

2.1.2 Patterns

The first widely acknowledged work on patterns in design was a book about the architecture of buildings by Alexander [17]. This book started at a very grand scale, for example showing how nations should be broken down into regions, and regions into smaller communities. The patterns progressed through to detailed studies. How to layout shelving is one example. The desire to abstract the built environment into a set of patterns may stem from Alexander’s inclination towards mathematics. His thesis refers to a set of programs he wrote in assembly code.

Object Oriented Patterns

In the field of software engineering patterns provide an opportunity to express and communicate repeatable solutions to common problems. The first examples of such patterns are attributed to Gamma, Helm, Johnson & Vlissides [62]. They are a group of software architects who aimed to create a set of design patterns for object-oriented software systems. This work inspired the creation of object-oriented patterns in more focussed domains [34, 132, 59].

Workflow Patterns

The workflow patterns are an effort to capture commonly encountered business process modelling scenarios [7, 151]. These patterns have been used to express what is required in the domain of business process modelling and what a process modelling tool should be capable of conveniently expressing.

The workflow patterns publications also represent an attempt to provide a comparison instrument for alternative workflow systems. So in that sense
they represent careful examination of the respective capabilities of many alternatives.

Patterns of Enterprise Integration

Hohpe and Woolf authored a comprehensive set of Enterprise Application Integration (EAI) patterns [77]. EAI is concerned with integrating core knowledge systems, legacy applications, and new/future applications within an enterprise. These patterns are basically a broad expression of what problems are encountered in this field and how to code solutions using middleware APIs. The EAI patterns build on previous work by Fowler in the field of enterprise application architecture [60].

This thesis proposes a comprehensive set of process integration patterns. These patterns significantly build on foundational concepts as introduced by Hohpe et. al. The process integration patterns in this thesis are higher level, and hopefully require less technical expertise to implement and understand.

Service Interaction Patterns

Thirteen service interaction patterns are presented by Barros et. al. [26]. The first seven patterns are essentially fundamental in nature (e.g. send, receive, one-to-many send, racing incoming messages). They are extremely coarse-grained. For example the pattern ‘one-to-many send’ doesn’t distinguish between the scenario where the sender addresses each recipient separately (multicast) and the scenario where a message gets sent to an abstract address that each receiver privately registers interest in (publish-subscribe). The final six patterns capture some interesting scenarios around process integration. They include contingent requests (starting a fallback interaction in the case of an unresponsive first interaction), and dynamic routing (passing a message along dynamic pathways, based on message contents). These patterns are interesting and appealing, however they are too fine-grained. They could easily be described as compositions of more fundamental and familiar integration patterns. For instance, the pattern ‘contingent request’ could be composed from a send-receive and a configurable exception flow, where the exception is a timeout. Another instance is the pattern ‘dynamic routing’, which could be a composition of channel passing patterns (wherein a channel reference is packaged in a message and the receiver learns about that channel from the message and uses it).
Correlation Patterns

The correlation patterns [25] capture many correlation scenarios. They are interesting, insightful and possess technical merit. BPEL implementations of the patterns are shown, which is possible in nearly all cases. The patterns in this paper are conceptually aligned with some of the proposals around correlation in this thesis. This thesis adds several new correlation patterns and shows how they can all be supported in a proposed language for process integration.

2.2 Distributed Systems Integration

The concepts of distributed systems integration are aligned with fundamental concepts of process integration. Consequently we will briefly examine them in this section. Furthermore the technologies of distributed systems integration (middleware) are frequently used as physical enablers of process integration. Indeed, distributed systems integration underpins process integration both conceptually and in terms of execution capability.

2.2.1 Trends

In the field of distributed systems integration a succession of integration technology movements, each markedly different from predecessors, have emerged. Their differences, in some cases, are counter-reactions to the perceived shortcomings of previous trends.

Sockets

Sockets were one of the earliest programmatic abstractions for enabling “inter-process” communication over networks [126]. They enabled connecting with a particular Internet computer and a particular application on that computer. Sockets were being pioneered during the early seventies [150] as a means of connecting processes on the ARPANET (the predecessor of the Internet).

Sockets allow message passing between network-connected computers. Sockets originally only supported uni-directional messaging, however later implementations allow bi-directional messaging and many forms of blocking and non-blocking communication. Sockets do not allow message connections to be typed.

Berkeley Sockets [146] came to be used widely during the early nineties. They are effectively an API abstraction of the TPC/IP Internet stack, and
are a foundational technology of the Internet.

**Distributed Objects**

In the early nineties Distributed Object (DO) technologies represented a shift towards an Object-Oriented approach to “inter-process” communication. Implementations of DO technologies include Microsoft’s DCOM [44] and CORBA [109]. DO technologies made objects on remote systems appear like they were local. They effectively hid marshalling and un-marshalling of data, distributed systems latency, and network location from the distributed objects that they integrated. The method signatures of remote systems enabled strongly-typed interface definitions. By allowing different remote methods to be invoked-by-name remote interface definitions became possible. The ability to know about exceptions that occur in remote objects meant that local applications could easily handle these errors and take effective remedial action.

To the architects of the client-server, and 3-tier systems of that time DO represented less programming effort than using sockets to integrate applications. Nevertheless, for some integration scenarios the remoteness of the integrated application was still too visible. In these cases the underlying concern was that the thread of control was typically shared between distributed applications (i.e. a synchronous architecture).

DO technologies were criticised for the fact that many of the executable implementations were proprietary, that they were overly complex, and that they created tightly-coupled architectures [72, 43].

**Message-Oriented Middleware**

Message-Oriented Middleware (MOM) technologies, such as the Java Messaging Service (JMS) [70], and Microsoft Message Queues (MSMQ) came into prominence after DO. By employing a document-oriented movement of data between applications and by decoupling the sender from the receiver MOM effectively overcame the tightly-coupled issues experienced by developers using DO.

Enterprise Application Integration (EAI) effectively used MOM technologies to integrate enterprise systems according to the tenets of loosely-coupled architectures. In that respect EAI could be construed as a strong counter reaction to the ‘mistakes’ of distributed objects. DO technologies were tightly-coupled and MOM technologies are loosely-coupled. Where DO made remote objects appear to be local, MOM made the integration feel like document transfer. Where DO forced the requesting application to wait for a
response/fault, MOM allowed the sender to fire-and-forget the interaction. This loosely-coupled paradigm was not the “silver bullet” for integration. New DO technologies are being continually created (e.g. Distributed Ruby). Despite some compelling reasons to loosely couple systems, problems requiring real-time or mission critical services make for some compelling reasons to tightly couple them.

Interestingly, around the same era, the Message Passing Interface (MPI) [48] employed a similar (but richer) message passing paradigm, for supporting computer clustering and supercomputers.

Service-Oriented Architectures

Service-Oriented Architectures (SOA) enable the creation of complex enterprise solutions from sets of enterprise services. SOAs represent some innovation. For instance SOAs can support loosely-coupled and tightly-coupled integrations. Nevertheless, most SOA instances (deployments) employ tightly-coupled SOAP/HTTP.

SOAs, unlike earlier approaches to integration, offer a means of integrating distributed applications without demanding identical programming languages/middleware infrastructures at each endpoint. This represents a significantly lower barrier to adoption. SOA, leverages Web service technologies such as SOAP [107], XML [32], WSDL [41]. By contrast, integration using CORBA required an Interface Definition Language (IDL) interface compiled into skeleton code and stub codes, and a common Object Request Broker middleware. SOAP does not necessarily demand so much infrastructure.

However SOAP is so open, and so extensible that it can be actually made platform/language dependent. For example .Net services can be exposed over SOAP using proprietary .Net datatypes and NTLM authentication protocols. This basically makes these services extremely difficult to contact from non .Net clients [42].

SOAs integrations are more model-driven than integrations using previous approaches. They allow users to quickly model and deploy aggregations of many Web-services into a composite service. SOAs offer control over the sequence of services being combined and the circumstances under which they are combined via languages for Web-service orchestration. Standards for SOA such as BPEL [18] and WS-Choreography [89] provide a conceptual framework for expressing how the component services are composed in an abstract manner. Such compositions (orchestrations) are typically modelled as a business process. The process “activities” are generally integration touch-points with the Web services composing the orchestration.
Enterprise Service Bus

Enterprise Service Buses (ESBs) are an emerging, evolutionary middleware technology [39]. The Enterprise Service Bus (ESB) is analogous to the local, hardware bus, carrying data to and from the CPU. The ESB, carries messages, in parallel, between applications. ESB, as a paradigm, combines the strengths of EAI (i.e. filtering, routing, and loose-coupling) and SOA (i.e. fewer technology barriers, Web-service standards, open architecture, and timely request-response).
ESBs offer many positive attributes in common with MOM:

**Mediated communication.** ESBs act as a mediator between distributed systems. They allow two communicating endpoints to communicate without the endpoints needing to know about each other.

**High volume communication.** The ability to support high transaction volumes is expected of an ESB.

**High reliability.** The ESB ought to be a highly reliable and highly available transit for enterprise messages.

ESBs offer many positive attributes in common with SOAs:

**Web-services enablement.** ESBs enable the creation and consumption of Web-services.

**Service Composition.** Like SOA technologies ESBs enable large numbers of smaller services to be composed into larger orchestrations using modelling tools.

**Real time communication.** Gartner uses the term the Real Time Enterprise (RTE) as a descriptive term for immediate, responsive connections between applications etc. [63]. Zero-latency messaging is a technology-based enabler of this real-time enterprise vision. ESB technologies are expected to offer some form of zero-latency messaging. This paradigm is at odds with the loosely-coupled paradigm asserted by proponents of EAI. Indeed the RTE paradigm represents a shift in thinking back towards the request-response style offered in Distributed Objects [75].

Additionally ESBs offer:

**Business rule based routing.** An ESB is able to route messages to a particular application based on their contents.

**Complex event processing.** If event \( A \) happens the ESB should do action \( X \), but if event \( A \) and event \( B \) happen within the same timeframe, the ESB should perform action \( Y \). Such complex event processing is feasible using code and a quality middleware, however the ESB allows it to expressed *codelessly*.

**Management and Tracking.** ESBs typically support the ability to keep tracking of interactions between enterprise resources for various purposes including auditing, reliability, and regulatory compliance.
ESBs have been described as representing little progress beyond SOAs and EAI (e.g. Microsoft [100]). Indeed, like SOAs, ESBs don’t represent a major change in the architecture of integrations, however ease of use and faster delivery times is their contribution to integration.

One major driver in the uptake of SOAs was the opportunity to be technology agnostic for the first time, which opened access to heterogenous languages and applications. Once again ease of use and technology independence are driving interest and uptake of ESBs through their earnest creation of off-the-shelf adapters. There is an ever smaller need to write code, because ESBs and their adapters make integration more model-driven. Off-the-shelf adapters have effectively replaced the APIs of Distributed Objects and Message-Oriented Middleware, and the Web-service descriptions of SOAs. The adapters are powerful and are typically pre-built to connect with a particular back-office system. Through a turn-key approach to integration with back-office systems (e.g. SAP ERP, Siebel CRM) technical risk is lowered and development times are significantly shortened.

### 2.2.2 Standards

Standards for distributed systems integration are numerous (e.g. JMS [70], CORBA [109], IIOP, SOAP [107], EDI [58], SWIFT [140], SMTP, RMI [138], EJB [53], FTP, and the numerous WS-* standards). A selected set of these are described with reference to how they relate to our work.

**Message Passing Interface**

The Message Passing Interface (MPI) [134] is a message passing paradigm. Like the message passing approach taken in the EAI movement it supports asynchronous exchanges, however it can also support realtime integration styles. The MPI was proposed as a standard by a consortium of vendors, developers, and users. It has bindings in several languages including C, C++, and Fortran. In terms of our research the MPI presents some good abstractions for message passing, and some of these ideas have been used in our proposals (see Chapter 3). Nevertheless MPI is designed for programmers of parallel applications and clustered computers.

**Distributed Computing Environment - Remote Procedure Calls**

Distributed Computing Environment - Remote Procedure Calls (DCE-RPC) is a standard for synchronous procedure calls over networks. DCE-RPC is aligned with the Distributed Objects movement mentioned previously. It
was proposed by the Open Group\textsuperscript{2}. RPC is typically limited to point-to-point topologies, with exceptions such as “Group RPC” [148]. RPC is functionally aligned with distributed objects. As a consequence RPC, like DO technologies, strengths of RPC include: that they can make remote services appear to be local, they hide latency and network location from the invoker, and they report remote exceptions back to the local invoker, giving it the opportunity to handle the problem.

Our research includes a broad array of integration options. Some of these options include the ability to integrate processes in an RPC-like fashion, if desired. However unlike RPC our research also offers highly decoupled alternatives as well – the choices fit into a generalised, conceptual framework.

The simplicity, and high availability of RPC comes at the expense of tightly-coupled solutions, possibly suffering from low cohesion [153] and poor design [77].

### Remote Method Invocation

Java Remote Method Invocation (RMI) [138] is a Java-based standard supporting RPC. RMI is aligned with the Distributed Objects movement. RMI inherits the same set of problems as RPC, (i.e. tightly coupled, low cohesion) and adds a Java only barrier to uptake.

RMI did improve RPC by making the remoteness more explicit apparent. RMI allows for the developer to test and handle remote exceptions within the client code.

### Common Object Request Broker Architecture

Common Object Request Broker Architecture (CORBA) [109, 83] is a distributed objects standard. Object design is defined using the Interface Definition Language (IDL). CORBA/IDL is an open standard proposed by the Object Management Group [111].

Implementations of CORBA are generally not cross-vendor compatible, due to gaps in the standard. The standard is a design-by-committee effort with over a decade of refinements behind it. The conceptualisation of integration suffers from far too much complexity, redundancy and bloat, making implementations inconvenient to use by business and IT analysts [43, 125, 24, 77].

\textsuperscript{2}Open Group also known as the Open Software Foundation \url{http://www.opengroup.org/} is the standards body behind the Distributed Computing Environment, which includes Remote Procedure Call
Distributed Component Object Model

The Distributed Component Object Model (DCOM) is yet another distributed objects standard. It is formerly known as NetworkOLE [61] and is Microsoft’s distributed version of their Component Object Model. It eventually became a standard and (like CORBA and RMI) is based on DCE-RPC. It inherits a similar set of challenges to other distributed object standards.

Java Message Service

The JMS [70] is a J2EE based standard for exchanging messages between distributed, autonomous applications. JMS is highly aligned with the EAI movement mentioned earlier. It was proposed by Sun Microsystems as a generic way for Java programs to exchange messages. There are a number of different JMS platforms, including Sun GlassFish\textsuperscript{3}, Websphere MQ\textsuperscript{4}, Tibco\textsuperscript{5}, and Oracle WebLogic for example.

Simple Object Access Protocol, Web Service Description Language

SOAP [107] is a protocol for enabling XML messages to be exchanged in a generic way over Web services. WSDL [41, 30] is an XML based standard for describing Web services. WSDL-SOAP is technology independent. This independence is achieved through abstraction of integration concepts. However, not helping SOAP and WSDL are the proliferation of ancillary standards for SOAP enabling eventing, publish-subscribe, reliability etc. (the WS-* standards). The conceptual overlap of these standards is a major barrier to uptake [77]. Representational State Transfer (REST) [57] has been proposed as an alternative architectural style to SOAP [156] and although it is not a standard, its simplicity and ability to leverage existing Web technologies (as opposed to proposing redundant new ones) has influenced the SOAP standard.

Business Process Execution Language

The Business Process Execution Language (BPEL) [18], is a process-oriented SOA standard. It represents the merging of two earlier workflow languages

\begin{footnotesize}
\begin{itemize}
\item IBM Websphere MQ, formerly known as MQSeries [79].
\end{itemize}
\end{footnotesize}
XLANG [142] by the Microsoft Corporation, and the Web Services Flow Language (WSFL) [95] by IBM Corporation. In April 2003 BPEL was submitted as a candidate standard to the standards organisation called OASIS.\(^6\) BPEL provides high-level abstractions for B2B integration (e.g. the partnerLink, and correlationSet). Lower level abstractions for integration are conceptualised by binding BPEL activities onto WSDL elements. A BPEL process is able to be exposed as a Web service meaning that in addition to invoking Web-services, a BPEL process can be invoked by Web-clients.

BPEL was released in 2002 and while it was conceptually quite strong and revolutionary, it did suffer some setbacks. Firstly it took a number of years after the release before any vendor tools fully supported it [76]. Secondly, by the time vendor tools supported BPEL another non executable standard was gaining popularity with the user community (called BPMN).\(^7\)

2.2.3 Technical Challenges

No matter what approach/standard is chosen to help with systems integration there are a number of technical issues that need to be addressed.

EAI Challenges

Hohpe [77] identified many common issues concerning message and interaction integrity. These technical issues are common to enterprise-grade distributed systems integration. A non-exhaustive list is included here:

**Lost messages** are problematic if they contain essential information. If lost messages are a problem, a great deal of modelling effort is required to ensure that lost messages are detected and resent. Imagine what happens if some e-payments never arrived? Personal experience demonstrates what can happen if an important email was sent but not received. Read-receipts in emails were created for this reason.

**Corrupted messages** the message may not be lost but the content may be altered en route, either deliberately or accidentally.

**Misrouted messages** messages may be received at the destination application but mishandled - or poorly routed from there. In this case such a message could be shunted to a dead letter channel, or just discarded.

\(^6\)“Organization for the Advancement of Structured Information Standards” http://www.oasis-open.org

\(^7\)BPMN [112] is a non-executable process modelling language.
by mistake. Or it could have been incorrectly addressed in the first place.

**Duplicate messages** that bring about state change may cause problems if acted on twice. How does the receive *know* not to act on an accidental second copy?

**Feedback** effectively reduces, or removes, uncertainty about the impact of an interaction. In mission-critical scenarios feedback may not only be essential, but must be immediate. The meaning of a feedback can take several forms but generally it signifies that (1) a message arrived at its intended destination, (2) the message was processed successfully, or (3) the following change has occurred as a result of the incoming message.

**Failures** can occur for any number of reasons and may need to be reported back to the sender. In other words a failure that occurred in service $x$ may influence services $y$ and $z$. Failures are a form of feedback and the actions taken as a result of a remote failure differ markedly from those taken otherwise. There needs to be a way of modelling this difference.

**Network drop-outs** are a reality in networks, and must be planned for. What happens if networks are down when a message is sent?

**Remote application crashes** can bring about a dis-alignment of state between two connected applications. Recovery procedures may be needed that will bring the two systems back into alignment.

**Message delivery latency** can be an issue for data having a short lifespan. How do systems differentiate messages containing stale data from those containing fresh data? How soon does it become stale? Is this time-length constant or does it change on a per message basis?

Each of these issues can lead to uncertainty about the state of the communicating partners. Without addressing/preventing each of these issues the sender cannot be certain about the integrity of an interaction. Likewise the receiver cannot be certain about the integrity of its response. Consequently two remote applications may be uncertain about the state of their counterparts, and be forced to undergo measures before proceeding with a conjoint action.

**Web-service Challenges**
**Web-service Standards Proliferation**  
Web-services offer a blessing and a curse to integrating distributed business processes. Their blessing is their obvious ties to the standard Web-protocols. Web-services exploit pervasive Web technologies. This in theory, overcomes software and platform heterogeneity, which is a major issue for EAI. By reusing the very Internet communication technologies employed by the standard Web browser (HTTP, port 80) Web-service technologies enable connections across tightly managed organisational boundaries through network proxies and firewalls.

One curse of Web-services is the result of the fact that HTTP was never originally designed to transport distributed systems integration messaging. That was one of the driving forces behind the proliferation of WS-* standards. For example, in order to standardize publish-subscribe over the Web the WS-Eventing standard created by the W3C was authored by members from Microsoft, IBM, Tibco, BEA [29]. But there is another set of WS-* standards for publish-subscribe communication over the Web (WS-Topics, WS-BaseNotification and WS-BrokeredNotification [145]) created by OASIS and authored by members from HP and IBM. Note that IBM is a co-author of both competing publish-subscribe standards.

The WS-Reliable Messaging standard [49] defines a protocol that enables reliable transfer of messages between Web-services. Web-services were originally intended to be a technology-agnostic, lightweight alternative to the sophisticated EAI middleware. However the issues being addressed in the WS-* standards are a counterforce to this original vision.

**Web-services Description Challenges**  
WSDL [41] is today the most commonly accepted way of describing any Web Service. It is supported (to some degree) by most Web-service development tools available. The alignment with the Web and its widespread vendor support make WSDL the most prolific technique for describing Services on the Internet.

The most recent version of the standard is WSDL 2.0, however WS-BPEL is aligned with its less powerful predecessor WSDL 1.1. It is intended to refine some of the less intuitive aspects of its predecessor and provide an improved and more powerful conceptual model.

WSDL also tends to place a surprisingly large emphasis on the style of binding, i.e. ‘encoded’ versus ‘literal’; and ‘RPC’ versus ‘document’. Butek [37] describes these as:

- **RPC/encoded** Inject the operation name into SOAP body and include a type attribute with every parameter value in the SOAP body.

- **RPC/literal** Inject the operation name into SOAP body and don’t type-define any parameter value in the SOAP body.
**Document/encoded** Inject no operation name into the SOAP body and include a type attribute with every parameter value in the SOAP body.

**Document/literal** Inject no operation name into the SOAP body and don’t type-define any parameter value in the SOAP body.

**Document/literal (wrapped)** Same as document/literal except this is a convention where the parameters are defined as being wrapped by the operation name – yielding the same SOAP structure as RPC/literal.

*RPC-style* and *Document-style* are two alternative ways of specifying SOAP messages. In RPC-style the client must inject the operation name into the SOAP body. In Document-style the structure of the SOAP message is defined fully in the message-part schema. *Encoded* injects type attributes, along with the data into the SOAP messages whereas *Literal* has no type attributes.

According to Butek developers struggle to understand the difference between these different styles of WSDL. The differences are subtle and minor between ‘RPC’, ‘Document’, ‘Literal’ and ‘Encoded’ and to some extent are really just about data. Any format of SOAP message borne out of these styles can be easily replicated using a Document/Literal style.

Interaction styles however, are not well defined in WSDL. Interaction styles are strongly related to the middleware technology being specified in the WSDL. WSDL documents must be extended with non WSDL information to bind onto any middleware transport technology. This is an intentional abstraction. However, it is a disjointed conceptualisation that prevents WSDL from locking onto the essence of an interaction. While it would appear to make WSDL more technology agnostic the abstraction creates a vacuum of information about how to bind services onto alternative middleware technologies. Consequently WSDL, in practice, nearly always binds onto SOAP/HTTP.

### 2.3 Process Integration

This section provides a brief overview of selected standards, research efforts, and software platforms for supporting process integration. Table 2.1 presents a set of process integration offerings. Strong support for a given category is denoted with a ‘+’. Moderate support and limited support are denoted with ‘+/–’ and ‘–’ respectively. Note that these evaluations are based on summary impressions of the respective offering’s support with respect to the solution criteria, as discussed in Section 1.2.3.
There is a diverse array of process integration standards. Process integration standards, like most standards are competing for the attention of the user community.

**Wf-XML**

The WfMC proposed a business process interoperability standard Wf-XML [139], which extends the Asynchronous Service Access Protocol (ASAP) [125]. Wf-XML and ASAP are protocols for process-to-process integration. They do not address the domain of modelling process integrations around these protocols, but there is an implicit, primitive meta-model. Wf-XML and ASAP, being protocols, rule out the possibility of technology-agnostic process integration. This is a significant barrier to adoption and is contrary to the objectives of this research.

**Business Process Execution Language**

BPEL is a highly relevant standard for process integration. It is regarded as a Web service orchestration language [119] (as opposed to a choreography language, which will be discussed shortly). As a standard for process integration standards, like most standards are competing for the attention of the user community.

Table 2.1: Comparison of process integration solutions.

<table>
<thead>
<tr>
<th>Offering</th>
<th>Expressive</th>
<th>Precise</th>
<th>Conceptual</th>
<th>Suitable</th>
<th>Understandable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workflow XML (Wf-XML)</td>
<td>+/-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Business Process Execution Language (BPEL)</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>BPEL-Lite</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Let’s Dance</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Business Process Modelling Notation (BPMN)</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Web Service Choreography Description Language (WS-CDL)</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Message Sequence Charts (MSCs)</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+/-</td>
<td>+</td>
</tr>
<tr>
<td>Rosetta Net</td>
<td>+/-</td>
<td>+</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Electronic Data Interchange (EDI)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td>Society for Worldwide Interbank Financial Telecommunication</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Activity Diagrams</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Harmonized Messaging</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B2B Integration Architecture</td>
<td>+/-</td>
<td>+/-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BPELChor</td>
<td>+/-</td>
<td>+/-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>REST-based workflow</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td>NINOS</td>
<td>+/-</td>
<td>+</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
</tr>
<tr>
<td>Cross-Organisational Workflows</td>
<td>-</td>
<td>+/-</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Enterprise Service Bus (ESB)</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+/-</td>
</tr>
</tbody>
</table>
integration BPEL is one of the conceptually stronger offerings. It allows the modelling of application integration, and WSDL’s extensibility model allows binding with heterogeneous middleware technologies. It offers concepts for correlating sets of interactions as if they belong to the same longstanding conversation.

Nevertheless, BPEL does have some weaknesses and they are significant:

- BPEL does not support human workflow and so it cannot, natively, model integrations between workflows containing manual tasks [9].

- BPEL is totally reliant on the Web services stack. As discussed in Section 2.2.3 there are significant conceptual flaws in WSDL. Consequently BPEL inherits all of these.

- BPEL cannot model integrations that cannot be described in WSDL (for example REST integrations [118]).

- BPEL is modelled in a set of XML documents, using multiple namespaces. This is definitely not suitable for humans to read or write.

BPEL has been discussed extensively previously in terms of its support for SOA (see Section 2.2.2). BPEL was also previously discussed in Section 1.1.2.

**BPEL Lite**

BPEL Light [108] proposes to use the extensibility features of BPEL 2.0 to dispense with the built-in communication activities and replace them with one new communication activity. The new communication activity is customisable and allows the interaction model to be defined independently of the process model. The motivation is that BPEL’s communication activities rely too heavily on WSDL bindings. The WSDL in a BPEL process is used to define the interfaces of partner processes, the access methods to integration middleware and message structures. By avoiding the use of WSDL, BPEL light focusses on the process structure the abstract interaction pattern, and the message structure.

Nitzsche et. al. argues that an *invoke* could be a combination of two asynchronous messages or a synchronous request-response. The differences are only relevant to middleware integrators [108]. However abandoning the conceptual details of integration make models incomplete and unbalanced. This is a violation of the 100 Percent Principle. Many core interaction concepts available in BPEL have been abandoned without being replaced in BPEL Light. For example message-to-process instance correlation, and partner linkages must be handled using code.
Let’s Dance

Let’s Dance [154] is a choreography language. As such it conceptualises a choreography as a non linear sequence of interactions. In Let’s Dance it is possible to model what interactions can occur before or after other interactions as well as what interactions must occur before or after others. In this respect the language represents an alternative (and extremely abstract) conceptual paradigm. This highly abstract idea has influenced a study on what interaction sequences can be enforced by the modeller on interacting Petri nets [52].

Business Process Modelling Notation

An initial (Beta 1) version of BPMN 2.0 [112] was release in 2009 (this is a draft version of BPMN 2.0). From this draft release it is clear that BPMN 2.0 includes extensions to BPMN 1.1 covering collaboration, conversations and choreography. The core concepts of the proposal are extremely abstract and high-level, making them appealing and perhaps compelling. The proposals appear to draw on several preceding components of work in the choreography field [52, 154]. However none of these proposals (including BPMN) include fundamental integration concepts typically used in executable process integrations. They all assert a box and arrows conceptualisation of integration.

There are a number of conceptual issues with BPMN:

- The ability to consume messages based on their contents is missing (message filtering).
- It appears not possible to express a type constraint over a communication pathway. Such a feature is necessary for abstract interface definition.
- There is an ItemDefinition metatype for defining the types of messages and events in BPMN. The metatype language is XML Schema, ruling out non-XML messages or events.
- Correlation keys appear as first-class elements on the proposal, however there is no concept for defining how correlation key values can be extracted from inside a message. i.e. there are no conceptual hooks between correlation keys and an ItemDefinition.
- There appears to be no conceptual handle for batch messaging groups of ItemDefinition elements.
• The ‘choreography task’ captures sender and receiver in the one task. The model assumption is that every message is delivered perfectly, only once. This is not a safe assumption with integrations. Unfortunately, there is usually real-world distances between the sender and receiver, and real world challenges that should not be ignored at any level of abstraction (e.g. messages get lost, delayed, or delivered more than once).

• There is no conceptualisation over the speed of message delivery.

• It is ambiguous whether communicating endpoints share control of execution at any moment in the interaction or not.

• There is no expression for the scenario where communicating endpoints can still interact despite large periods of being disengaged with the communication medium (i.e. offline). This is a core capability of JMS, for example. The extreme case of this is where both endpoints are never ‘online’ concurrently and yet can exchange messages.

• Time can pass between the moment messages are sent and the moment they get received, causing messages to become stale. BPMN offers no concept of message expiry.

• BPMN provides only an abstract (not concrete) construct for expressing the location of an endpoint.

• A differentiation between publish-subscribe and multicast appears missing. The difference in what the sender needs to know and do is significant. For instance publish-subscribe requires the sender to know about a topic, whereas multicast requires the sender to know the location of every recipient and to interact with each individually.

• The ability to model synchronous or asynchronous interactions is not supported. Synchronous interactions support real-time scenarios and asynchronous interactions support massively-parallel scenarios. Highly different scenarios should not be indistinguishable when modelled.

BPMN as a process integration language abstracts from middleware concepts by avoiding them. Thus it is going to difficult for models to be made executable. Integration fundamentals such as interaction design, message structure, access methods and locations possibly form 70 - 80% of the effort in any integration project yet they are missing from BPMN. The 100 Percent Principle would require these to have some form of presence.
In process integration the way (how) messages get delivered, the location to which (where) they get delivered, and the rate of transfer (speed) at which they are delivered are of paramount importance. These notions cannot be expressed in BPMN.

In favour of BPMN is its strong high-level conceptualisation of process integration. Traditional methods of distributed systems integration are typically steeped in technical terminology and are generally avoided by all but the most technically minded people. This proposal appears to be one of the most compelling languages for non technical business analysts to use to start talking about integration. It may provide the right combination of elements to BPM modellers whose primary interest is creating non-executable documentation.

**Web Service Choreography Languages**

Web service choreography languages enable modelling of collaborations between many distributed processes. Choreography languages include Web Services Choreography Interface (WSCI) \[19\] and the Web Services - Choreography Description Language (WS-CDL) \[20, 33, 89\].

Choreography languages, in general, provide a means for describing complex, longstanding sets of interactions between many participant processes. The model essentially takes a *birds-eye* perspective of all the participant processes from the standpoint of what they communicate. No participant process owns/manages the choreography \[119\].

Choreography languages typically offer *some* support for expressing control flow and conditional-routing on any participant process. It is possible to express data types for messages.

BPEL, and the WS-Choreography languages are highly integrated with XML standards \[32, 107, 41, 56\]. Nevertheless they only integrate with services using the Web service stack. The Web service stack has several, overlapping standards for eventing, security, reliability etc. (as mentioned in Section 2.2.3). This proliferation is confusing and unnecessarily complex for end-users \[57, 77\].

This research differs from Web service composition languages (e.g. BPEL) and Web-service choreography languages (e.g. WS-CDL and WSCI) in several ways. Firstly we aim to provide alternatives to Web service bindings in our integration. Secondly, our techniques are founded in formal methods.

---

8WSCI was originally prepared by Intalio, Sun Microsystems, SAP, and BEA. WSCI, being a Web service orchestration language (like BPEL), builds on WSDL. WSCI was submitted as a “technical note” to the W3C.

9WS-CDL is a replacement to WSCI by the W3C.
such as Petri nets and ORM. Also our proposals will lead to extensions of an executable workflow language.

**Message Sequence Charts**

Message Sequence Charts (MSC) [65, 84] are a language for describing real-time interactions between distributed components. MSCs are mostly used for modelling telephony switching systems. They have high level graphical abstractions, can model events, messages, instances, inheritance, data, time and can even model state. As MSCs matured a formal semantics was created, and they became standardised. There are plenty of good ideas in this set of standards. However MSCs are not a competing proposal, as there is little potential for their use in business process integration. Furthermore they appear to lack concepts for space decoupling and time decoupling (see Chapter 3).

**RosettaNet**

RosettaNet [127] defines a set of standards for XML/EDI that relate to supply-chain relationships between electronics and semiconductor manufacturers. They capture e-business scenarios such as a “request for quote” (PIP 3A1), or a “request for shipping” (PIP 3B12). They are firmly focussed on their specific B2B domain.

**Electronic Data Interchange**

Electronic Data Interchange (EDI) is a B2B standard. For over twenty five years EDI\(^\text{10}\) has been a viable technology for business-to-business integration [35, 36]. This standard has not managed to make much impact. In practice EDI was marginally adopted by large enterprises in specialised domains [66]. EDI, in its role as an integration standard, has inherent problems such as vendor lock-in, general inflexibility, and insufficient process awareness. It does not support complex data types [88].

**SWIFT**

Society for Worldwide Interbank Financial Telecommunication (SWIFT) is another B2B standard. It is specifically focussed on the needs of the financial sector. SWIFT has enjoyed widespread adoption in Europe for over ten years and continues to be widely used.

\(^{10}\text{EDI - Electronic Data Interchange}\)
This research will focus on integration problems, as they relate to business processes. This research will be more generalised in its design and the problems it addresses.

**Activity Diagrams**

UML Activity Diagrams [113] are a graphical based modelling language for representing workflows. They are published as part of the UML 2.0 standard by the OMG. Activity Diagrams, being part of UML, are conceptually linked to “objects”. They can represent the control flow, data, and even the resource perspective of workflow. They can also capture exception handling semantics. Activity diagrams do not seem to explicitly handle messaging; nor are they considered to be executable. When the OMG acquired BPMN it basically abandoned research and development around Activity Diagrams.

**Harmonized Messaging**

In a proposal by Sadiq & Orlowska [129, 115] some extensions over and above those offered by state of the art SOA/EAI platforms are proposed to enhance the integration capabilities of business processes. The main aspects of these extensions include sequencing, correlation, batching, filtering, and temporal messaging. The work is interesting and has many alignments in ideas and purpose with this thesis. This thesis goes beyond [129, 115] by offering a more thorough investigation of concepts underlying process integration and also offers a language, formal syntax and early implementation.

**B2B Integration Architecture**

The work of Bussler [36, 35] provides a highly engineered, sophisticated architecture for business-to-business (B2B) integration. Bussler’s architecture proposed interface processes to expose processes to other businesses. Interface processes are virtual. They are really just a facade to real internal processes existing within the organisation. However the proposal is just an architecture, and does not propose a language or a platform on which to execute the language. Nevertheless, for its time (2003) the architecture is insightful and worthwhile.

**BPEL4Chor**

BPEL4Chor proposes an XML wrapper around two or more BPEL documents, along with a process topology describing messageLinks between each process and a grounding describing how the topology relates to a WSDL
portType/operation [50]. BPEL4Chor, is a set of seemingly minor extensions to BPELs syntax. It inherits BPELs strengths, as well as its weaknesses (see Sections 2.3.1 and 2.2.2).

What is not clear from this proposal is the conceptual redundancy of BPEL4Chor over plain BPEL. In plain BPEL co-operating processes can be modelled to perform complementary roles over the the one WSDL interface. The BPEL partnerLink and partnerLinkType constructs enable roles to be defined over a service. Thus two BPEL models could constitute a global perspective over a choreography without further definition. Despite this BPEL4Chor offers a messageLink construction (inside its topology and grounding constructs) which appear to basically capture the same information as the mandatory BPEL partnerLink. This apparent redundancy enables creation of inconsistent (self-contradicting) integration models.

The Web-enabled BPEL4Chor modeller tool built on Flash provides the ability to model BPEL4Chor graphically. BPEL correlationSets are not yet shown in the graphical syntax. This would be a great improvement to the editor.

**CorPN**

Zhao et al. [155] propose an approach (referred to as Correspondence Petri Nets – CorPN) for process integration where focus is on process instance correlation. The approach builds on high-level Petri Nets in order to define collaborating process models. Nets are extended with communication places and cardinality constraints put on arcs, capturing the idea that process instances of one process can relate to one or many process instances of another process. In the proposed approach, processes need to be linked at design time and there is therefore no provision for emerging communication relationships between processes (rather than process instances). In this thesis there is no mandate for messages to track correlating process instances like there appears to be in CorPN. The approach in this thesis also enables the creation of abstract, data-based rules for consuming messages and thus not only focuses on the issue of process instance correlation. Also, the paper does not discuss an implementation nor how the proposed approach could be aligned with state of the art middleware solutions.

**REST-based workflow**

Zur Muehlen et. al. [156] proposed a REST-based [57] approach to process integration. Their proposal identifies each process instance as an Internet resource (URI). This allowed for process instances to essentially own and
share their URI with remote collaborative processes. When each process instance uses its own URI it is trivial to make sure that the right process instance is matched to an incoming message.\footnote{Traditional integration approaches expect all process instances to share the same communication channel. This raises the issue of how to ensure that messages are routed to the right process instance.}

BPEL for REST [118] applies a REST-based approach to process integration. The Web methods (\texttt{get, put, post} and \texttt{delete}) are all proposed to have explicit meaning when communicating with a particular process instance.

Each of these proposals has technical merit and leverages the successful architecture underlying the Web. One of the core ideas in REST is that every instance of a resource is given its own unique URL. This way stateful service instances can be easily distinguished from one another on the Web. This principle has shaped an important (one of many) message correlation technique proposed in this thesis.

**Cross-Organisational Workflows**

An architecture to enable cross-organisational workflow was proposed by Schulz and Orlowska [133]. Their architecture is based around the idea of creating a mediator process that contains only the public workflow steps that both collaborating parties wish to share and work together on. Each participant process executes a hidden (private) workflow while sharing a coalition workflow. The coalition workflow details actions performed in both processes. This idea is very similar to the Interface Process presented by Bussler [36]. The proposal presented contains a great deal of technical detail and careful presentation. The work does not concern itself with integration issues per se, and overlooks the possibility of simple control flow structures (such as XOR splits) in the coalition workflow. The coalition workflow would have advantages in several specific integration situations, however it would also add complexity and would raise questions about control and ownership.

**NIÑOS**

NIÑOS [97] is an agent-based, service-oriented workflow architecture. The agents each execute atomic segments of the workflow. They are loosely-coupled with one-another. Whereas traditional workflow orchestration engines are centrally managed by one application NIÑOS distributes the burden of process execution through the agents, which can execute in different locations. Each agent is designed to execute a single activity in the workflow.
Agents are able to forward scheduled work to each other over a content-based publish/subscribe infrastructure (PADRES [96]). This architecture allows agents that are being over utilized (bottlenecking the execution) to be multiplied without interrupting the process flow. The architecture is highly scalable and adaptable to varying server loads during runtime. The architecture is highly contrasted to this thesis in terms of problems being solved. The proposal seeks to address the problems of handling huge volumes of instances through a distributed grid of workflow agents. The location of deployment for each agent is highly flexible. This thesis, on the other hand, aims to speed up the development speed of integrated processes irrespective of innate heterogeneity of process execution platforms and heterogeneous integration middleware. Perhaps the strengths of this thesis are the weaknesses of NINOS, and vice-versa. It would be interesting to investigate if the two approaches to process integration could be combined such that the strengths of each complement one another.

2.3.2 Software

The BPM market is characterised by users who have already invested (sometimes significantly) in enterprise software. There are literally hundreds of BPM and Workflow tools available along with thousands of makes and models of enterprise systems. Arguably the greatest challenge experienced by users in the BPM market is the need to integrate processes with heterogeneous enterprise systems. Consequently the vendors are focussed on providing as many adapters as possible.

Some BPM platforms are discussed in this section with respect to their process integration capabilities.

Tibco+Staffware

Tibco [144] offers a comprehensive suite of integration tools. Tibco is a mature integration platform that has descended from the mainframe systems integration domain. Strategically, when Tibco acquired Staffware they, sought to bolster their offering with the Workflow/BPM capabilities offered by Staffware.

Staffware workflow system supported rudimentary ‘out-of-the-box’ integration with Microsoft desktop applications (using OLE and DDE) and supported integration with enterprise applications by delegating this task to a third-party integration processing server. Connections to remote applications was designed and performed using ad-hoc techniques and divergent API calls [136].
Staffware is no longer available however Tibco is now a combined workflow/ESB product. It can trigger workflows using remote events and can listen for combinations of events etc. However Tibco appears to employ a bottom-up conceptualisation of integration – causing it to feel a little ad hoc. This thesis builds on many ideas also found in Tibco, including composite eventing, high-speed interactions, batch filtering and conversation modelling. Furthermore it grounds these integration fundamentals into an over-arching conceptual framework.

**Websphere MQ Workflow**

Websphere MQ Workflow is IBM’s BPM platform. Websphere MQ Workflow allows integration with applications using RMI-IIOP as well as the message queuing facility of MQSeries, however the modelling technique for each of these integration styles is completely different despite their similar intent and action.

Websphere MQ Workflow can invoke applications using turn-key components. These integration services are tightly bound to IBM’s z/OS platform, and some proprietary integration protocols [80]. Websphere MQ provides several APIs supporting process-to-application integration using C, C++, COBOL, ActiveX, Java, Messaging and SOAP.

Websphere MQ Workflow [80] is able to perform most application integration scenarios, however its highly engineered architecture, and complex APIs make it extremely difficult to use and learn by non experts. The level of modelling/coding effort required to perform even basic integration is high, when compared with other BPM offerings.

**SAP NetWeaver**

SAP NetWeaver is a service-oriented integrated application platform for many of SAP’s business products. NetWeaver comes with tooling for process integration. These include:

- SAP Composite Application Framework (CAF)\(^ {12}\) which enables a model-driven approach to creating new applications from compositions of finer grained SAP services.

- SAP Process Integration (PI)\(^ {13}\) provides ESB and SOA services on the SAP Netweaver platform.


\(^ {13}\)SAP Process Integration is formerly called SAP eXchange Infrastructure [137].
SAP CAF can enable creation of new SAP applications with minimal coding, while Process Integration is able to support huge transaction volumes. SAP PI enables the creation of interfaces from external applications into SAP’s many services (e.g. IDOC, ABAP etc.) and vice-versa.

Nevertheless SAP PI and SAP CAF are two discrete tools. When they need to interoperate they need to be integrated – even though they are both SAP integration tools. SAP PI enables integration with the various components of SAP however it is not particularly suited to integration with non-SAP processes. This is due to complex conceptualisation and a lack of turn-key integration adapters to external middleware/process software. Nevertheless there are third-party ESB offerings that offer turn-key integration between SAP and external applications. For example iWay B2B\(^{14}\) and webMethods\(^{15}\) are ESB platforms that offer turnkey integration between SAP non-SAP applications. These third party ESB platforms are good at accessing the SAP RFC APIs and SAP IDOC interfaces directly without the need for SAP BI.

**BPM Integration Plug-ins**

Enterprise friendly BPM solutions generally include turn-key plug-ins and components for connecting to both well-known middleware and remote applications (for instance Salesforce\(^{16}\) and Interwoven\(^{17}\)). These plug-ins serve the same logical function (and offer the same advantages) as the adaptors for ESB platforms mentioned previously.

**2.4 Conclusion**

Business process integration is a field of endeavour that is being actively developed in both academia and industry. Generally speaking, these efforts could be grouped into those that appear to start at the highest levels of abstraction and work down (*top-down languages*) and those that appear to start with something executable and built on it towards higher levels of abstraction (*bottom-up languages*).


\(^{16}\)Salesforce: [www.salesforce.com](http://www.salesforce.com) accessed January 2009

\(^{17}\)Interwoven: [www.interwoven.com](http://www.interwoven.com) accessed January 2009
Top-down offerings (e.g. BPMN, π-calculus, Harmonized Messaging, and B2B Integration Architecture) are generally conceptually clean and unified. However, if a top-down language doesn’t capture any nuances of integration technology, it cannot easily be made executable. An executable top-down language should capture the nuances of alternative real world middleware, otherwise the 100 Percent Principle would be violated. Most top-down languages are non-executable.

Bottom-up offerings (e.g. SAP Process Integration, Websphere Workflow MQ, and BPEL4Chor) start with something executable, and build additional layers onto it. Generally they are grounded in integration technology and are easily executed. If a bottom-up language is not carefully built, however, they can become ad hoc. The concepts can be technology-driven leading to undesirable duplication of the same fundamental concept.
Chapter 3

The Dimensions of Loosely/Tightly-Coupled Architectures

3.1 Introduction

Modern approaches to integrating distributed systems almost invariably rely on middleware. These middleware take many forms, ranging from distributed object brokers, to message-oriented middleware and enterprise service buses. Each family of middleware is designed to provide proven solutions to a certain set of distributed system integration issues, but how can one compare their relative strengths and weaknesses when they are so different at the core? Indeed, a formally defined conceptual framework for integration middleware remains elusive.

The heterogeneity of contemporary middleware reflects the absence of a consensus on the right set of communication abstractions to integrate distributed applications [27]. This lack of consensus can be observed even for the most elementary communication primitives, namely send and receive. For example in sending a message, the semantics of what happens (a) if it doesn’t arrive, or (b) if it gets buffered (and if so if this happens locally/remotely); is usually not clear until the choice a technology has been made. As noted by Cypher & Leu: “the interactions between the different properties of the send and receive primitives can be extremely complex, and ... the precise semantics of these primitives are not well understood” [48]. Coupling, or the way in which endpoints are connected, lies at the heart of this issue.

The style of coupling can vary from one family of middleware to another. For instance solutions inspired by CORBA [109] use the Remote Procedure
Call (RPC) paradigm, wherein the sender and receiver applications typically become synchronised at the thread level. Conversely Message-Oriented Middleware (MOM) solutions use what is usually referred to as an asynchronous approach. In spite of these differences there are MOM implementations supporting request-response, e.g. Java Message Service (JMS) [70], and there are RPC solutions supporting asynchronous interactions. While this sounds encouraging it doesn’t mean that these solutions are interchangeable at the conceptual level. Consequently, conceptual comparisons become mired in technical detail. It is one of the goals of this chapter to decouple conceptual comparisons from technical ones - at least when it comes to inter-coupling.

A full analysis of middleware would be a daunting task. The set of features is large, particularly when one considers, for example privacy, non-repudiation, transactions, time-outs, and reliability. This chapter focuses on the notion of (de-) coupling, as it is the source of many distinctions central to the design of distributed applications. Specifically, the chapter formulates a framework for characterising levels of coupling. Novel contributions of this chapter to the conceptualisation of integration include:

- A detailed analysis of the notion of decoupling in middleware and a formal semantics of dimensions of coupling in terms of Coloured Petri Nets (CPNs) [87].

- A collection of notational elements for integration modelling based on the notions of coupling previously identified. These notational elements are given a visual syntax extending that of Message Sequence Charts (MSCs) [128].

- A classification of middleware in terms of their support for various forms of (de-)coupling. This classification can be used as an instrument to assist in middleware selection, although requirements outside the scope of this study may play an equally important role in middleware selection (e.g. transaction support, non-repudiation and message filtering capabilities).

- A prototype middleware demonstrating the possibility of supporting all styles of coupling in a consistent and uniform way.

This chapter is structured as follows. Section 3.2 establishes a nomenclature. Section 3.3 formalises a set of coupling dimensions while Section 3.4 shows how these dimensions can be composed, leading to a set of unidirectional coupling integration patterns that provide a basis for integration modelling. Section 3.5 introduces bi-directional interactions and incorporates
them into the same conceptual framework. Section 3.6 presents a framework for comparing middleware solutions based on the uni-directional, and bi-directional patterns. Section 3.7 introduces JCoupling: an open source prototype that combines all of the proposals of this chapter. Sections 3.8 and 3.9 present related work and conclusions respectively.

This chapter is based on previously published analyses of coupling for elementary interactions [11, 15]. The latter extends the former, incorporating publish-subscribe techniques into the models of interaction, and by encompassing compound interactions (e.g. request-response). We have introduced remote fault reporting, timeouts, aggregate messaging, and multicast - making the proposals more grounded. The latter publication also introduces a new proof of concept prototype (JCoupling) that implements the theoretical models presented in both publications. The latter publication is a refinement of the ideas presented in our previous technical report [14].

This chapter does not discuss a range of modelling elements that would allow us to model an integrated process. Nor, does it discuss an implementation that solves process integration. What it does is define and frame the problems of process integration. This way we can understand what a solution must be able to do in order to support the modelling, and execution of integrated process models. This way we can measure and evaluate some of the ‘solutions’ based on their real merits rather than the claims of the authors.

In this chapter we introduce some of the technical and conceptual, challenges and hurdles to the problems of process integration. These challenges are presented as patterns, or requirements or enablers for integrated processes. In this presentation we have attempted to not only deal with requirements that are unique to process integration, but also have attempted to tie these notions into commonly understood requirements for enterprise application integration.

3.2 Background

This section defines key terms related to coupling used in the rest of the thesis.

A message is a discrete unit of information (containing for example a command, data, request, or an event) that is passed between endpoints. Depending on the technology it may contain header metadata (e.g. message ID/ timestamp) and/or payload data. It may be transactional, reliable, and may be transported over a “channel”, or even a TCP socket.

An endpoint is a communicating entity and is able to perform interac-
tions. It may have the sole capability of sending/receiving messages and defer processing any information to something else, or it may be able to communicate and process.

An interaction refers to endpoints exchanging information [122]. The most basic interaction is a uni-directional message exchange (elementary interaction).

A channel is an abstraction of a message destination. This abstraction is similar to the notion of “queue” supported by JMS, WebsphereMQ [80], and Microsoft Message Queue (MSMQ) [104]. A common attribute of common queues (channels) offered by WebsphereMQ and MSMQ is that they are not necessarily private. For instance, these solutions permit many endpoints to invoke a receive request over the same queue (channel) concurrently. In such a case the middleware treats these two endpoints as if they are in fact competing for a message [77].\(^1\) Channels can be extended with many functions. Examples include preservation of message sequence [55, 48], authentication, and non-repudiation [77].

A private channel is slightly different to a channel in that in this case there is something preventing more than one receiving endpoint from being able to compete for messages on the same channel. A private queue (available from WebsphereMQ), is one example where the middleware explicitly doesn’t allow certain channels to be shared. Conversely in cases the design or architecture of the middleware won’t allow shared channels by design; for example TPC Sockets don’t support many applications sharing the same IP-address/port combination.

A topic is another form of symbolic destination. Like channels many receivers may consume messages off one topic – the difference being that all receivers get a copy of the message.

Eugster’s survey of Publish-Subscribe [55], introduced three dimensions of decoupling:

- **Thread Decoupling** – (referred to by Eugster as Synchronisation Decoupling) wherein the thread inside an endpoint does not have to wait (block) for another endpoint to be in the ‘ready’ state before message exchange begins.

- **Time Decoupling** – wherein the sender and receiver of a message do not need to be involved in the interaction at the same time.

- **Space Decoupling** – wherein the messages are directed to a particular symbolic address (channel) and not directly to the address of an

\(^1\)In such circumstances one endpoint will get the message and the other will typically be made to continue waiting/timeout.
The dimensions of decoupling have relevance to a large spectrum of middleware, including MOM, space-based [64], and RPC-based middleware.

3.3 Distributed Systems Coupling

This chapter is based on Eugster’s dimensions of coupling. In this section we present conceptual representations and Coloured Petri nets (CPNs) for coupling distributed applications. At this stage we focus on uni-directional interactions only. Bi-directional interactions (e.g. request-response) are presented in Section 3.5.

We chose CPNs to formalize the proposed notions of coupling. While we could have used other models of concurrent systems, e.g. Process Algebra [73] or \( \pi \)-calculus [102], we preferred CPN due to its graphical nature, its mature (and freely available) tool support, and the fact that the concept of buffer, which is central in the formalisation, maps directly to the concept of place in Petri nets.

All CPNs have been fully implemented and tested using CPN Tools [45].

3.3.1 The Thread-coupling Dimension

Thread-decoupling enables “non-blocking communication”, for either, or both, the sender and receiver. Non-blocking communication allows the endpoint’s thread to interleave processing with communication. In the following paragraphs we introduce some notational elements for various forms of thread decoupling as well as the CPN formalisation.

Pattern 1: Blocking Send

A message send action can either be blocking or non-blocking. A blocking send implies that the sending application must yield its thread of control while the message is being transferred out of the local application. It does not matter if it is passing the message over a wide area network connection or to another local application. If the sender’s thread, at least, blocks until the message has left the local application and its embedded middleware, it is blocking\(^2\). Figure 3.1(b) is a CPN of a blocking send. The outer dashed line represents the endpoint while the inner dashed line represents middleware client code that is under control of the endpoint. These do not form part of the CPN language and are used only to indicate architectural concerns.

\(^2\)If the interaction was bi-directional it would block longer (Section 3.5).
After initialising a send action, the transition “process” cannot fire until a thread is returned at the end of message transmission.

The next two paragraphs introduce the CPN notation [87]. In Figure 3.1(b) the circular nodes are called ‘places’ (e.g. “msg ready”, “app contrl”, and “in progrs”), and they represent potential states. The rectangular nodes are called ‘transitions’ (e.g. “initiate send”, “begin x-port”, and “process”), and change a CPN from one state to another. Tokens, e.g. the black dots in the places “msg ready” and “app contrl”, represent a particular state of the model. Figure 3.1(b) is in a state where a message is ready to be sent, and the application has control of its own thread. This is the initial state of the CPN, and it is referred to as its ‘initial marking’. According to the firing rules of CPNs two transitions (“initiate send” and “process”) are concurrently enabled for the initial marking. When either of these transitions fire it will consume one token from each of its ‘input places’ and produce one token into each of its output places. A transition is enabled if every one of its input places contains at least one token. For example, firing “initiate send”, will consume one token from each input place. Consequently the transition “process” becomes disabled because “app contrl” now contains zero tokens. Transition “process” cannot become re-enabled until the transition “finish x-port” returns a token to the place “app contrl”. “Finish x-port” only returns this token once the message has left the sender. In Figure 3.1(b) the places are type-constrained. For instance the place “msg ready” can only hold tokens of type Msg. This is shown as an annotation to its bottom-right side. The annotations shown at the top-right side of the places “msg ready” and “app contrl” are references to constant values. The values are used to determine the initial marking of Figure 3.1(b). For instance the annotation “msg1” is a constant value of type Msg. Its presence puts a token into the

3By contrast, firing “process” will remove the token from “app contrl” and then place a token back in the input place (because it has a bi-directional arrow).
place “msg ready”. Each arc entering/leaving a place in this CPN is annotated by a variable, which is typed according to the place the arc connects to. Arcs leaving transitions specify the value of the token to be produced and may use arbitrary ML [103] functions. This feature will be used in Section 3.3.3.

In Figure 3.1(b), when a message is ready (represented by a token inside the place “msg-ready”) and the application is ready (represented by a token inside the place “app-control”) the endpoint gives the message to the embedded middleware. The endpoint yields its thread of control to the embedded middleware, getting control back once the message has completely left the embedded middleware. Inside the embedded middleware the transitions “begin-x-port”, “fin-x-port”, and the place “in-progress” are placed over the edge of the endpoint. This denotes that the remote system (receiver endpoint or middleware service) will bind to the sender by sharing these transitions and the place. The assumption is that inside the middleware, at a deeper layer of abstraction, systems communicate in a time-coupled, thread-coupled manner, regardless of the behaviour exposed to the endpoint applications. Therefore this CPN may be “transition bounded” with remote systems.

In a blocking send there is a thread coupling of the sender application (endpoint) with something else – but not necessarily the receiver as we will show in Section 3.3.2.

Pattern 2: Non-blocking Send

A thread decoupling is observable at the sender in the case of a non-blocking send [55]. A non-blocking send means that message transmission and local computation can be interleaved [134]. Figure 3.2(a) presents a notation for non-blocking send, based on the MSC notation [128]. Figure 3.2(b) defines the concept in CPN form. This figure, like that of blocking send (Figure 3.1) is transition bounded with remote components through the transitions in the

---

4In this series of CPNs we represent tokens as black dots. This is not strictly necessary as the initial markings are shown textually. It is however, a convention we adopt that is intended to assist their readability.

5Not all underlying network protocols are time-coupled and thread-coupled (e.g. User Datagram Protocol), however if a messaging technology is going to offer any degree of reliability or data integrity a time-coupled, thread-coupled technique becomes essential. For instance WebLogic JMS (BEA WebLogic www.bea.com accessed March 2008), and Email/SMTP (http://www.ietf.org/rfc/rfc0821.txt accessed June 2008) both use Transmission Control Protocol (which happens to be thread-coupled and time-coupled); despite this fact they both expose thread/time decoupled APIs.

6“Transition bounded”, in this context, means that two distributed components share a transition (action), and must perform it at exactly the same moment.
embedded middleware of the application. Snir and Otto provide a detailed description of non-blocking send [134].

![Diagram of non-blocking send](image)

Figure 3.2: Non-blocking send. The transition “process” can be interleaved with communication steps because a thread is not yielded to the embedded middleware.

A non-blocking send is a necessary condition, but not a sufficient condition to achieve total thread decoupling, which is to say that the receive action must also be non-blocking. Rephrased, if both the send and receive are blocking (non-blocking) then a total thread coupling (decoupling) occurs. A partial thread decoupling occurs when the send is blocking and the receive non-blocking, or vice-versa.

The non-blocking send is fairly uncommon for middleware solutions. For instance all RPC-based implementations use blocking send. MOM implementations such as “Websphere MQ” [80] and MSMQ partially support it; but only if the middleware service was deployed onto the local host, the send operation only blocks while the message is passed between applications on the same machine. However, this is not always possible, or practical due to licensing limitations, or the limited computational power of a small device. We propose that such middleware could be improved by exposing an explicit non-blocking send in the API.

**Pattern 3: Blocking Receive**

Like message send, message receipt can either be blocking or non-blocking [48]. The definition of blocking receive is that the application must put a thread into a waiting state in order to receive the message (ownership of the thread is usually returned to the endpoint when the message is received). This means that the receiver thread is coupled to either the message sender
directly or to some form of queue on a middleware service (depending on whether the middleware queues messages between the sender and receiver, or not). Section 3.3.2 formalises this notion. Figure 3.3 presents a notation and model for blocking receive. When the transition “initiate receive” of Figure 3.3(b) is fired, the token sitting in place “thread” is consumed, and a token is put into place “rcv ready”. Consequently, the transition “process” is no longer enabled. Meaning that the model cannot progress until a message arrives (i.e. the receiver is in the blocking state). This may occur before a corresponding sender endpoint sends its message.

We assume fairness in the firing of non-deterministic transitions. In other words, when two transitions are enabled simultaneously – such as “initiate receive” and “process” in Figure 3.3(b) – a random choice is made between them with equal probability that either transition is taken. This random choice simulates a race condition. The assumption of fairness is common in the field of Petri nets and it is reflected in the way in which Petri net simulation and analysis tools operate [87]

In the CPN language a technique known as transition guards can be used to form a basis for selection between two concurrently enabled transitions. Transition guards are Boolean expressions that operate over the token data, only allowing a transition to fire when true. However in this case, we did not use transition guards. When two concurrently enabled transitions are modelled without transition guards (such as “initiate receive” and “process” in Figure 3.2(b)) the CPN makes no assertion about which path gets taken. The semantics, in such a case, are that an unknown random influence will determine which transition gets taken. Obviously, in real systems this choice is not random, and is probably determined by application logic. Nevertheless our decision to model this selection as being random simplifies our model, but does not but does not alter the semantics at this particular level of abstraction.

**Pattern 4: Non-blocking Receive**

The non-blocking receive occurs when the application can receive a message, without forcing the current thread to wait. This is illustrated in Figure 3.4.

A well-known embodiment of non-blocking-receive is the event-based handler described in JMS [70]. Once a handler is registered with the middleware it is called-back when a message arrives. The Message Passing Interface (MPI) provides another embodiment of non-blocking receive that is not event-based [134], wherein the receiver polls for the presence of a new message.

Non-blocking receive, as a communication abstraction, seems less popular
than blocking receive. This is probably because blocking receive interactions are simpler to program and debug [77]. One frequently observes statements about the merits of an asynchronous architecture. While such statements are indeed valid, they usually refer to the merits of queuing messages between the sender and the receiver, and have little to do with the fact that the threads of control are blocked or not. Therefore the threading dimension as presented in this section is orthogonal to the general usage of the word “asynchronous”.

Figure 3.3: *Blocking receive.* A thread must be yielded to the embedded middleware until the message has arrived.

Figure 3.4: *Non-blocking receive.* A thread need not be yielded to the middleware in order to receive.
3.3.2 Time

The dimension of time decoupling is crucial to understanding the difference between many peer-to-peer middleware paradigms and server-oriented paradigms (e.g. MPI versus MOM). In any elementary interaction time is either coupled or decoupled.

**Pattern 5: Time-Coupled**

*Time-coupled* interactions, like those typically found in MPI, cannot take place unless both endpoints are concurrently connected. There is no possibility of queueing messages between the two endpoints. In time-coupled arrangements the interaction begins with the message being wholly contained at the sender endpoint. The transition-boundedness of endpoints can guarantee that the moment the sender begins sending the message, the receiver begins receiving. This concept is presented in Figure 3.5 wherein the endpoint applications are joined directly at the bounding transitions (“*begin x-port*” and “*finish x-port*”). Time-coupled interactions accord with the general usage of the word “synchronous”. Figure 3.5 does not show the endpoints. The “sends” and “receives” may be blocking or non-blocking. Hence, Figure 3.5 should be seen in conjunction with the earlier figures.

![Figure 3.5: Time coupling is characterised by transition-bound systems.](image)

**Pattern 6: Time-Decoupled**

*Time-decoupled* interactions allow messages to be exchanged irrespective of whether or not each endpoint is concurrently operational. In this case messages are queued between the sender and the receiver. To achieve time decoupling a third endpoint is needed, that both the sender and receiver can...
access. Time-decoupling is presented in Figure 3.6. The two endpoints, and the middleware service are again transition-bounded. However now, the middleware service is able to buffer the message, as captured by the place “buffr’d”.

The place “buffr’d” is filled with a token when the sender has finished sending a message, and the receiver has not started receiving that message. One token in the place “buffr’d” represents one message being held on a middleware that sits between the sender endpoint and the receiver endpoint. In Figure 3.6 the proceeding transition will fire immediately because, in this CPN the presence of a token in “buffr’d” enables it. Like in WebLogic’s JMS implementation there is (conceptually) no limitation on the number of messages that can be buffered.

![Diagram](image)

(a) Notation, time decoupled.

![Diagram](image)

(b) CPN of time decoupled messaging.

Figure 3.6: *Time decoupling* is characterised by the presence of an intermediate endpoint.

MOM solutions such as Websphere MQ and MSMQ are typically deployed in a “hub and spoke” arrangement, which is truly time-decoupled. An alternative arrangement is the “peer-to-peer” topology. In such a topology the sender endpoint and middleware service are deployed on the same host. This latter topology may seem time-decoupled, but it is not because interactions can only take place if both hosts are concurrently connected to the network.
This may be a problem for endpoints on hosts on unreliable networks, e.g. mobile devices.

### 3.3.3 Space

Space is the final dimension of decoupling. By “space” we refer to a dimension of coupling that takes into account the degree to which sender endpoint can control which instance receives the message. If the sender can totally control which endpoint instance would receive the message we can assume a high degree of space coupling, and vice versa.

**Pattern 7: Space Coupled**

A space coupled interaction has the property that only one receiver can possibly receive the message. This behaviour could be driven by, for example, hard location data (e.g. an IP-Address/port combination), or more abstract mechanisms incorporated into a middleware (e.g. private queues in IBM Websphere MQ). Figure 3.7 presents the concept of space coupling. The CPN for space coupling is presented at the end of this section.

**Pattern 8: Space Decoupled (Channel) and Pattern 9: Space Decoupled (Topic)**

Space decoupled interactions on the other hand allow a sender to send a message without requiring explicit knowledge of the receiver’s address. Decoupling in space generally makes architectures more flexible, and extensible.

There are two distinct forms of space-decoupling. Space-decoupled architectures permit one sender to interact with *one of many* receivers (over a channel) or *many of many* receivers (over a topic), without uniquely identifying the receiver. Figure 3.8 introduces extensions to MSCs that present notations for “channel” and “topic” based interactions.
(a) Channel - one receiver gets the message.

(b) Topic - all receivers get the message.

Figure 3.8: Extensions to MSCs representing two forms of space decoupling.

Figure 3.9: CPN presenting a semantics for the space dimension. The transition "begin x-port" has been annotated with a box labelled "Allocate", which means that this transition decomposes to a sub-net (see Figure 3.10).

In Figure 3.9, the transition "begin x-port" decomposes to the sub-net shown in Figure 3.10. The sub-net models the behaviour of the three options of the space dimension illustrated by figures 3.7, 3.8(a), and 3.8(b). The places "new subsc", "subscriptions" and transition "subscribe" model the ability for receiver endpoints to subscribe to destinations. Each receiver has its own unique application ID (type Apid). Hence a new subscription token (i.e. "apID, dest") is an endpoint/application ID combined with the relevant destination. Transition "subscribe" takes this token and appends it to the list of subscriptions represented by the token in place "subscriptions". This is performed in the following ML expression (apID, dest)::subs. The double colon :: (an append operation) adds the data element (apID,dest) to the head of the list subs.

Sub-nets can be used to hide details. Specifically, a transition can be decomposed into a sub-net and by replacing this "substitution transition" by its decomposition one obtains its semantics.
There are three I/O places ("init’d", "rcv ready", and "in progrs" etc.) common to parent and sub-net. These places can also be found in the CPNs of the Thread-coupling, and Time-coupling dimensions. By linking with these I/O places a sender can expect interaction to occur according to one of the three patterns of space-coupling – depending on the type of the destination. The sender pushes a token into the place “init’d” (which is typed DestMsg). Likewise, a receiver, linking to this model can push a token into the place “rcv ready”. This place is typed ApidDest and it identifies this receiver’s application and the channel, or destination, over which the message is to be received.

Figure 3.10: This CPN is the ‘Allocate’ sub-net refining transition “begin x-port” in Figure 3.9.

Figure 3.10 presents the sub-net decomposition of the transition “begin x-port” (from Figure 3.9). This CPN shows how a unified underlying model could perform any of the three options within the space dimension (i.e. topic, address, or channel). From this we can conclude that it is possible to offer a clean implementation to all three forms of space (de-)coupling within the one middleware solution.

The input places ("init’d" and “rcv ready”), the output place (“in progrs”), and the input/output place (“subscriptions”) correspond to the similarly labelled places of the parent net (Figure 3.9). Note that a subscription is essentially a combination of a reference to a destination (of type topic) and a backward reference to the subscribing application. Similarly, an address essentially consists of a destination reference (of type address) being bound to one listening application. This similarity explains the fact that transition “begin addr” shares the input place “subscriptions” with transition “begin
Firing transitions “begin addr” or “begin chan” require the appropriately typed input tokens, and produce one token referring to the message and its intended application. The difference between the two is that “begin addr” can guarantee its token output will always identify the same application for any input address, whereas “begin chan” cannot guarantee which application “wins” the message. Firing the transition “begin topic” only requires an input token identifying a topic-message, then the application IDs in the subscription set “subs” (a set of <application-ID, destination> pairs) that are subscribed to that topic effectively are allocated a copy of the input message. Each subscriber can then obtain, or peel off, its copy of the message using the same technique as for addresses and channels, i.e. by putting a token into the input place “rcv ready”. Hence while the interface for each of the three options of space coupling are consistent, the way messages get bound to application-IDs is different.

Thus, direct addressing supports interactions between a sender and only one receiver. Channels send the message to one of many receivers, and topics to all of many receivers. We have chosen not to add a conceptual element that encompasses the notion of a many-to-many interaction. Despite the fact they are observable in real world systems, they seem to be better described as combinations of one-to-one/one-to-many interactions described in this chapter; and/or many-to-one interactions [13].

Patterns M1 & M2 - Multicast and Message Joining

Multicast involves sending a message to many receivers. This is orthogonal to the space dimension, and in particular, publish-subscribe. Indeed multicast corresponds to a set of interactions to several destinations. It can be achieved by performing several interactions in parallel or in any order. Each of these interactions may have as a destination, an address, an channel, or a topic. Such an approach yields Pattern M1.

The converse to multicast is the multiple message join (Pattern M2) or aggregate receival. Essentially many messages are received as an aggregated batch. Once again, this is orthogonal to the dimensions of (de-) coupling.

Support for multicast and message join are useful features of a messaging solution. Thus they are strongly related, however they do not cut into the coupling dimensions.

---

8Shown in Figure 3.9, is the transition for adding a new subscription to either an address or a topic. In this a transition guard prevents more than one application binding to an address at any time, while allowing many applications to subscribe to the same topic.
3.3.4 Summary

The coupling integration patterns defined in this section consolidate existing intuitive notions of tightly versus loosely-coupled communication. Having conceptualized these notions, we can now reason about them and about their possible combinations. In particular, we can seek to confirm the orthogonality of these notions. Moreover, the coupling integration patterns can help to delineate fundamental differences between various forms of middleware. This opens the potential for using the dimensions as a middleware selection instrument.

We contend that this set of coupling integration patterns can be used in defining interaction requirements during system analysis and design. Each pattern is sufficiently specific and precise in terms of concept and behaviour. They are sufficiently different that some will be more suitable to a given integration problem than others depending on the systemic requirements.

The dimensions of decoupling included “thread decoupling” (with its four options), “time decoupling”, and “space decoupling” (with its three options). Each dimension has its own precise behaviour and semantics, and were introduced in a graphical notation based on MSCs, and more precisely as CPNs. The CPNs shared similar structure and place names, which provides a hint of how to combine these dimensions together, while preserving their individual semantics.

3.4 Combining Threading, Time, and Space

We contend that the dimensions of decoupling presented in the previous section are orthogonal to each other. Hence, any pattern for thread-coupling can be combined with any pattern for time-coupling, which in turn can be combined with any pattern for space-coupling. Hence, a quick combinatorial calculation yields twenty four \((2^2 \times 2 \times 3 = 24)\) possibilities for uni-directional coupling. We use the term coupling integration configuration to designate a given combination of options across the three dimensions. Figures 3.13, 3.14, and 3.15, in Section 3.4.2, enumerate a merged notation for each combination. The semantics of each coupling integration configuration can be precisely defined by a CPN, obtained by combining selected CPNs for each of the three dimensions as explained below.

The set of achievable combinations, of these dimensions, can be used as a palette of ways to couple/decouple systems and can thus be applied to an integration problem or to the selection of an appropriate middleware product.
3.4.1 CPN Merging

By combining the CPNs for each of the three dimensions, using CPN Tools, we have verified that the dimensions are indeed orthogonal. In other words, any CPN from each of the dimensions can be combined with any CPN from any other dimension, and the resulting merged CPN preserves the behaviour of its constituent CPNs.

Furthermore the behaviour of the combined CPN still exhibits the behaviour implied by its constituent source CPNs. This means one can create integration models that are precise, and unambiguous thereby allowing a degree of strong separation between a model of integration, and the technologies used to achieve it.

Minor adjustments are required when merging the CPNs. Due to space constraints we will only present the procedure we followed to create two of the twenty four possible merged CPNs. The other twenty two possible CPNs can be created similarly.

Figure 3.11(b) models a thread-coupled (i.e. synchronisation-coupled), time-coupled, space-coupled combination. Our first step was to start with the blocking send (Figure 3.1(b)) and blocking receive (Figure 3.3(b)), and merge these two CPNs together along their similarly labelled nodes “begin x-port”, “in progrs” and “finish x-port”. This merged CPN is time-coupled, and therefore we do not add any intermediary between the blocking send and the blocking receive.

![Figure 3.11: Modelling a thread-coupled, time-coupled, space-coupled interaction.](image)

To produce Figure 3.11(b) we changed the types of selected places, and arcs to account for the space dimension. We specialised the type (from \textit{Msg} to \textit{DestMsg}) for the places “msg ready” and “init’d”, and likewise specialised the arc variables. We also specialised the type (from \textit{Thread} to \textit{ApidDest})
for the place “rcv ready” and inserted a new place “msg req”. A token in this place contains an identifier of the receiver (Apid) and the Channel/Topic/Address (Dest). The place labelled “thread” was specialised from Thread to Appid, and the places “msg in” and “in progrs” were specialised from Msg to ApidMsg. Finally, we added the sub-net “Allocate” (Figure 3.10) to the transition labelled “begin x-port” and linked the same transition to the place subscriptions.

A second example of a merged CPN is presented in Figure 3.12(b). It models a thread-decoupled, time-decoupled, space-decoupled (channel) interaction. Being time-decoupled we added a “middleware service” between the sender and receiver, and stitched the boundary nodes (transitions and places) of the sender and receiver to this service.

![Modelling a thread-decoupled, time-decoupled, and space-decoupled interaction.](image)

In Figure 3.12(b) we specialised the type of selected places of the sender and receiver using the same approach presented for creating Figure 3.11(b). The type for the places “buffr’d” and the sender side “in progrs” was also specialised (from Msg to DestMsg). Finally we added the sub-net “Allocate” to the transition labelled “begin x-port”. However, being time decoupled, the modification is only performed to the transition on the receiver side.
3.4.2 Composing the Dimensions Graphically - An Enumeration

To create a graphical representation of a coupling configuration the graphical notations associated with each of the nine coupling patterns presented in Section 3.3 can be used a starting point. They are, more or less, obtained by “overlaying” the notations for decoupling introduced in Section 3.3. As stated in Section 3.4, there are twenty-four possible combinations of these nine patterns, given that they represent points on different dimensions, or vectors. These graphical notations extend Message Sequence Charts [128].

![Diagram of coupling configurations](image)

Figure 3.13: Notations for the coupling integration configurations – space-coupled communication.
Figures 3.13, 3.14, and 3.15 present these combined notations.

Figure 3.14: Notations for the coupling integration configurations – Space-decoupled (Channel).

3.4.3 Procedure for building a combined CPN

In this section we present a general procedure that builds combined CPNs from the CPNs of the coupling dimension patterns presented in Section 3.3. The commonly named transitions and places from each dimension (e.g. “begin x-port” and “in progrs”) are the stitching points we use to build these combined CPNs.
Figure 3.15: Notations for the coupling integration configurations – Space-decoupled (Topic).

**Step 1** To the choice of sender - Fig. 3.1(b) or 3.2(b) - update the places “msg ready” and “init’d” from the colour-set $\text{Msg}$ to the colour-set $\text{DestMsg}$ (as shown for the space dimension CPN in Fig. 3.9). This combines Destination and Message data.

**Step 2** To the choice of receiver - Fig. 3.3(b) or 3.4(b) - update the places “rcv ready” and “msg req” to the colour-set $\text{ApidDest}$ (as shown for the space dimension CPN). This identifies the receiver application ID ($\text{Apid}$) and the Channel/Topic/Address ($\text{Dest}$).
Step 3 Join choice of sender - Fig. 3.1(b) or 3.2(b) - to the choice of receiver - Fig. 3.3(b) or 3.4(b) - using boundary nodes “begin xport”, “in progrs”, and “fin xport”. Include (exclude) the Middleware service in Fig. 3.6(b) if the combined model is (not) time-decoupled.

Step 4 Bind the sub-net “Allocate” (Fig. 3.10) to the transition “begin x-port”. If there are two so labelled transitions (as is the case in time-decoupled models) only bind the sub-net “Allocate” to the receiver’s instance of transition “begin x-port” as shown in Fig. 3.12(b). For future reference we’ll call this transition begin-x-port(Allocate).

Step 5 Update the places “in progrs” and “msg in” leading away from the transition begin-x-port(Allocate), from the colour-set Msg to the colour-set ApidMsg. This identifies the intended receiver application ID (Apid) and the Message.

Step 6 If the merged CPN is time-decoupled update the places (“in progrs” and “buff’d”) leading towards the transition begin-x-port(Allocate) from colour-set Msg to colour-set DestMsg as shown in Fig. 3.12(b).

In summary we have presented a method for binding a sender CPN to a receiver CPN, and an optional, intermediate buffer CPN (used if the systems are time-decoupled).

### 3.5 Bi-directional Interactions

Bi-directional interactions generally involve a requestor, and a respondent. As a general observation, middleware based on an RPC paradigm support bi-directional interactions, while Message-Oriented Middleware are oriented towards uni-directional messaging. For instance RPC-based middleware (e.g. CORBA [109]) supports “request-response” interactions, whereas MOM based middleware (e.g. Microsoft Message Queueing – MSMQ), do not natively support request-response interactions.

There are exceptions to this general observation, such as the JMS API. JMS provides partial support for request-response using its QueueRequestor and TopicRequestor classes, but these only serve to mask the two interactions occurring over the native JMS provider from the requestor’s perspective. Further, these classes do not report remote exceptions and the TopicRequestor returns only the first response and ignores the responses of all other subscribers.
The analysis of (de-) coupling has thus far only accounted for unidirectional interactions. Bi-directional interactions, the exchanging of data in two directions, occurs within the scope of a single interaction for RPC style middleware. We challenge the notion that only thread-coupled systems allow bi-directional messaging within an interaction.

Bi-directional messaging, adds additional possibilities for instance delivery receipt, and the reporting of receiver-side faults. A delivery receipt (i.e. system acknowledgement) is an event returned to the requestor, that its message has been successfully received. A delivery receipt (as distinguished from a response) does not imply that the targeted endpoint has processed the message, just that it has received it. Finally a receiver side fault being propagated back to the requestor indicates that an error occurred during the processing of the request message.

Likewise timeouts are also a relevant aspect to consider when modelling interactions. So far they have not been presented even though we have modelled application-level faults. Section 3.5.2 presents a timed-CPN of Blocking-send, Blocking-receive showing how timeouts fit into that particular pattern combination. Section 3.5.2 also summarises the results of simulations conducted on these timed CPNs.

Therefore users of time-decoupled solutions are typically forced to use work-arounds to implement request-response interactions. The designers of time-decoupled solutions appear to overlook these types of interactions despite the fact that they are an essential requirement for many forms of distributed computing.

The CPN models from Section 3.3 covering threading and time were extended to optionally support request-response interactions, fault notification, and/or delivery receipt. These extended CPNs (see figures 3.16 and 3.17) preserve the orthogonality of time, space, and threading. In Figure 3.16 the structure of the boundary nodes ("begin x-port", "in progrs", "finish x-port", "begin respnse", and "finish respnse") in either CPN is identical. The major difference being that Figure 3.16(a) waits for the result, whereas Figure 3.16(b) continues processing immediately. The application in Figure 3.16(b) can rendezvous with the result when it is ready.

One can observe that the alternative CPNs of Figure 3.17 do preserve their blocking and non-blocking behaviour respectively. The CPN for blocking receive (Figure 3.17(a)) includes the return of a delivery receipt, whereas non-blocking-receive (Figure 3.17(b)), includes the return of a response/fault. A delivery receipt is not intrinsic to blocking receive, just as responses and fault...

\(^9\)Note also that the prototype mentioned in Section 3.7 addresses timeouts and networking faults.
Figure 3.16: Extended CPNs dealing with thread-coupling in bi-directional interactions from the requestor side. The changes made to the related CPNs from Section 3.3 are black, while the unchanged elements are grey.

notification are not intrinsic to non-blocking-receive. They are presented in these CPNs as alternatives, a choice more inspired by expediency.

We have seen that it is necessary to extend the CPNs for thread-coupling in order to cover responses. It is also necessary to extend the CPNs for the dimension of time. Figure 3.18 (request, optional response, time-decoupled) shows that the response is generated by the intermediate point of the interaction. This means that for time-decoupled interactions the semantics that two systems can interact without being active concurrently is preserved. The place “resp ready” stores and returns a signal to the requestor indicating the message has been buffered, and is ready for the receiver to retrieve. If the case arises that the requestor still wishes to retrieve a response from the respondent in a time-decoupled manner, the CPN of Figure 3.18 allows for this by providing an optional response polling service. This is started at the place “polling” and continues through transition “bgn td respnse”. We do not include the extended CPN for “time-coupled” interactions, because it is a trivial extension of Figure 3.5(b).

We consider that the use of the response mechanism presented here should
be optional. Implementations that provide this range of integration services should not force users to use, or even retrieve responses. However it would seem sensible if implementations, like the CPNs consistently performed responses, and simply left it optional for the clients to retrieve them.

We have concluded that in fact delivery receipts (system acknowledgements), responses, and faults can be added to the semantics of blocking/non-blocking, time-coupled/time-decoupled topologies without interfering with their original semantics, as defined in Section 3.3. Therefore it would be useful, when comparing middleware solutions in terms of their coupling, to also take into account their relative support for alternative patterns of response.

3.5.1 Patterns of Response

Based on our survey of middleware solutions/standards, and theoretical modelling, we have arrived at these patterns of request-response:

**Pattern R1: Preprocess acknowledgement** a signal, provided by the middleware, is returned to the requestor indicating the successful receipt of the message.
Pattern R2: Postprocess acknowledgement  a signal, provided by the middleware, gets returned to the requestor indicating that the message was successfully processed.

Pattern R3: Postprocess response  a response, containing information provided by the respondent, gets returned to the requestor.

Pattern R4: Postprocess fault  an exception/fault occurs in the respondent while processing the message, and information about this gets propagated back to the requestor.

Pattern R5: Receive response: blocking  the requestor blocks for the response.

Pattern R6: Receive response: non-blocking  the requestor thread uses a non blocking technique to receive the response.
The options for responding to a message do not interfere with the semantics of coupling and decoupling. For example, time-decoupling enables, but does not force “fire and forget” interactions. Likewise, a blocking-send does not imply a response, and a non-blocking send does not imply the lack of one. The primitives of coupling and their formal semantics help clarify this orthogonality despite the support (or lack thereof) by solutions and standards.

3.5.2 Extended CPNs and Simulation

Figure 3.19 presents a combination of a Blocking-send, and a Blocking-receive CPN - extended - to include a timeout. Either the sender or the receiver can timeout before the interaction has completed. Some new places, transitions and arcs were added (indicated in Grey) to perform timeouts. Note that the structure of the original CPNs (Figs. 3.1(b) and 3.3(b)) has been preserved. Nevertheless, we extended the colours Thread and Msg to timed (see 1 in Fig. 3.19). We also extended some arcs with timed expressions (see 2). The timed expression msg@+m adds m time units to the token msg. Variable m is declared as a random number between 4 and 7 (colset M = int with 4..7). This range of values was chosen for simulation purposes in such a way that the timeout may or may not occur. When the token msg is timed any transition it enables will not be allowed to fire until the current time equals m. We also changed two arcs to carry multiple tokens (see 3).

Figure 3.19: Blocking-send, Blocking-receive with timeouts.

Change (3) was necessary because if a timeout occurred in the sender...
(removing a token from the place “in progr”) we need to clean up the incomplete interaction for the receiver (and vice versa). Put another way, if a timeout occurs during message transit for one endpoint the model must allow the other endpoint to timeout gracefully, preventing deadlock. Technically, using two tokens as shown by change (3) allows this.

Using CPN tools we ran 40 simulations of this CPN. We found that there were three possible outcomes. Each simulation run consisted of 50 transition firings starting with the initial marking shown in the CPN (two messages in place “msg ready”). After each simulation run we checked that: (i) the CPN was not in a deadlock; and (ii) the two messages had arrived or they were aborted due to a timeout. In other words, we tested that messages in the initial state always flow to a possible final state. It turned out that 50 transition firings was sufficient to ensure that the messages flowed from beginning to end so there was no need for longer simulation runs. The simulation engine of CPN tools implements a randomized transition firing algorithm that ensures fairness in the way transitions are fired during simulations.

**Timed-out, no interactions** before the transition “fin x-port” had the time to fire – completing an interaction – both of the transitions “Timeout send” or “Timeout recv” fired.

**Timed-out, some completed interactions** at least one interaction completed such that transition “fin x-port” fired.

**All completed interactions** before any time-out transition could fire, all interactions were completed.

The delay to fire a transition (that progresses an interaction) was produced using a random value in the range of \( m \) (e.g. \( \text{msg}\theta+m \)). This value
Figure 3.21: Subnet decomposition “Finish x-port” of the transition “fin x-port” in Figure 3.19.

influenced the interaction speed. The delay to fire either of the time-out transitions was produced using a random value in the range of $i$ (for our simulation runs $i$ had a range of 15 .. 20). The value taken by $i$ is used to establish the timeout time (e.g. $t@+i$). In our simulation runs we found that by increasing the interaction speed (i.e. decreasing the range of $m$) we increased the number of all completed interactions. Conversely, by increasing the timeout speed we increased the number of timed-out, no interactions. These results are exactly in line with our intuition.

3.6 Comparison of Middleware Solutions and Standards

The concepts presented in this chapter, are supported by a range of middleware solutions and standards, but to varying degrees and in different combinations. This can make the selection of a middleware solution quite difficult, especially considering that the language used in different middleware “camps” can be difficult to penetrate, due to subtle changes in meaning of common words.

Therefore, in this section we propose that these concepts can be used to evaluate various middleware solutions and standards. Such an evaluation may be of assistance to architects deciding between technologies based on the requirements with respect to levels of coupling or decoupling between distributed systems. To illustrate this proposition, we have evaluated the
following: Java-spaces\textsuperscript{10}, Axis \textsuperscript{[21]}, CORBA \textsuperscript{[109]}, JMS\textsuperscript{11}, Websphere MQ\textsuperscript{12}, MSMQ \textsuperscript{[104]}, and MPI\textsuperscript{13}.

First of all it is important to establish that with any of these standards and solutions, it is possible to implement all of the concepts. The question we are addressing is the relative level of effort required to achieve it. The results of this evaluation are presented in Table 3.1 in Section 3.6.1. Solutions that directly support a pattern are given a plus (‘+’). Those able to support a pattern using minor work-arounds are given a plus minus (‘+/–’), and those requiring greater effort are assigned a minus (‘−’). A detailed rationale behind these assessments is provided in Section 3.6.2.

### 3.6.1 Comparison Table

In Table 3.1 Patterns C1 to C9 in the table represent the coupling integration patterns presented in Sections 3.3.1, 3.3.2, and 3.3.3. Patterns M1 and M2 represent the two patterns related to multicast and message joining presented in Section 3.3.3. Patterns R1 to R6 represent patterns of request-response as presented in Section 3.5.

### 3.6.2 Rationale behind the evaluation of Standards and Tools against the Patterns

Table 3.1 presented an evaluation of middleware standards and tools, in terms of their ability to directly support or partially support the coupling capabilities presented in this chapter. Solutions that directly support a pattern were given a plus (‘+’). Those able to support a pattern using more than a little effort were given a plus minus (‘+/–’), and those requiring greater effort were assigned a minus (‘−’). A ‘−’ symbol, assigned to a standard/solution, does not mean that the achievement of this pattern is impossible; rather, the ‘work-arounds’ necessary to achieve the pattern, using this standard/solution, are non-trivial.

**Java Spaces**

Java Spaces supports a blocking-send through its \texttt{write} operation. However it does not support a non-blocking send, hence capability ‘2’ was given a ‘−’


\textsuperscript{12}Websphere MQ V 5.1, \footnotesize{[80]}.

\textsuperscript{13}MPI Core: V. 2, \footnotesize{[134]}.
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Option</th>
<th>#</th>
<th>Java Spaces</th>
<th>Axis</th>
<th>CORBA</th>
<th>JMS</th>
<th>Websphere MQ</th>
<th>MSMQ</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threading</td>
<td>Blocking Send</td>
<td>1</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Non Blocking Send</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Blocking Receive</td>
<td>3</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Non Blocking Receive</td>
<td>4</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Time</td>
<td>Time Coupled</td>
<td>5</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Time Decoupled</td>
<td>6</td>
<td>+</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Space</td>
<td>Space Coupled</td>
<td>7</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Space Dec. Channel</td>
<td>8</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Space Dec. Topic</td>
<td>9</td>
<td>+/-</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
<td>+/-</td>
<td>+/-</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multicast</td>
<td>Multicast/Scatter</td>
<td>M1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Joinmsgs/Gather</td>
<td>M2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Request-Response</td>
<td>PreprocessAck</td>
<td>R1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>PostProcessAck</td>
<td>R2</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+/-</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>PostProcessResp</td>
<td>R3</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+/-</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>PostProcessFault</td>
<td>R4</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Blocking Receive Ack</td>
<td>R5</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Non Blocking Receive Ack</td>
<td>R6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.1: Evaluation of selected middleware solutions and standards.

Java Spaces supports blocking-receive through its `read` or `take` operations, and supports non-blocking receive through its `notify` operation, hence capabilities ‘3’ and ‘4’ were assigned a ‘+’ in Table 3.1.

Java Spaces, is time-decoupled, and is not time-coupled. Consequently capability ‘5’ was assigned a ‘-’ in Table 3.1.

Java Spaces supports space-decoupling over a channel, but does not support coupling one sender to one receiver, hence a ‘-’ for capability ‘7’ in Table 3.1.

Space-decoupling over a topic (e.g. publish-subscribe) is partially achieved if each ‘subscriber’ uses a `read` operation. Each receiver gets a copy of the message because the call to `read` does not remove the message from the space. Hopefully the message will be removed from the space before any receiver reads the same message twice. A simple, but not fail-safe ‘work-around’ to prevent this problem is to write the message to the space with a very short lease.

84
Java Spaces does not provide primitives for sending messages over an arbitrary set of templates, and we therefore rule out multicast. We rule out support for message joining because it does not support its take, or read operations over arbitrary sets of templates.

Java Spaces does not directly support responses, therefore patterns R1 – R6 of Table 3.1 were assigned a ‘–’.

**Axis**

Axis is a SOAP engine that primarily uses HTTP as a transport. Like HTTP, it only offers a blocking-send, and a non-blocking-receive. Despite the fact that Axis can be configured to use JMS as a message transport service, it does not expose a non-blocking send or a blocking-receive in its API. Consequently non-blocking send and blocking receive were assigned a ‘–’ in Table 3.1.

Axis directly supports space-coupling, but does not support space-decoupling. Despite the fact that JMS supports space-decoupling over channels and topics Axis does not expose constructs that allow users to exploit either of these features. Consequently, in Table 3.1, items ‘8’ and ‘9’ were assigned ‘–’.

Axis provides no direct support for multicast, or for message joining.

Axis supports many form of acknowledgement, hence it was well represented in the list of items of Table 3.1 related to responses. Nevertheless, preprocess acknowledgement (R1) is not supported. A ‘work-around’ solution would require forking off a thread in the server to process the message while sending a receipt acknowledgement in the response from the main thread. Non-blocking receive of responses is not possible without using callbacks, hence it is assigned ‘–’ in Table 3.1.

**CORBA**

CORBA supports a blocking-send because the CORBA client blocks on remote object calls, at least until the request has reached the ORB. It supports non-blocking-receive because the remote object servicing requests never makes an explicit request to wait for an incoming request, this gets managed by the ORB. Nevertheless, CORBA provides no direct support for a non-blocking-send or a blocking-receive operation.

CORBA traditionally offers a time-coupled means of interacting, and using what it refers to as an ‘asynchronous’ style it provides direct support for time-decoupled interactions.
CORBA’s naming service is the primary way to address remote objects. The naming service maps a name to one remote object, as opposed to one of many, hence CORBA directly supports space-coupling. CORBA allows clients to obtain remote object references using Interoperable Object Group Reference (IOGR). This sort of remote object reference refers to one of many object implementations, and therefore CORBA supports space-decoupling over a channel. CORBA also has support for publish-subscribe, but achieving this style of interaction is not as straightforward as it should be, and therefore we consider that it only partially supports this (see Table 3.1).

CORBA does not directly support multicast or message joining hence we gave it ‘’ for M1 and M2 of Table 3.1.

CORBA, like Axis would require thread forking techniques to provide the requestor with an acknowledgement before the request gets processed, therefore we gave response pattern R1 a ‘’ in Table 3.1. Otherwise CORBA has the best representation of any of the solutions or standards evaluated for supporting the various forms of response, and that is reflected in Table 3.1.

Java Message Service

The JMS standard directly supports blocking-send, blocking-receive, and non-blocking receive. However, it does not directly support non-blocking send. Nevertheless, most implementations of JMS can be configured with a sender, and middleware service on the same machine. The message is passed into the middleware service and then is put over the network without the sender needing to wait. Hence we assign JMS a ‘+/–’ for item ‘2’ in Table 3.1.

JMS, being a MOM driven standard, natively supports time-decoupling but not time-coupling. Hence a ‘’ for item ‘5’.

JMS supports both forms of space-decoupling (channel and topic), but not space-coupling. Consequently item ‘7’ in Table 3.1 was assigned a ‘’.

JMS, however does not support either of the multicast/join patterns.

JMS is not a request-response driven standard, but despite this it supports preprocessor acknowledgements (R1) directly via session acknowledgements. It supports blocking-receive acknowledgements (R5) directly through its QueueRequestor and TopicRequestor classes. Post-process acknowledgements (R2) and post-process responses (R3) are supported through the same classes. However, we deem this support only partial (‘+/–’) due to the need to parse for a return address and begin a new interaction at the responder (as discussed in Section 3.5).
Websphere MQ

Websphere MQ is the MOM component of the Websphere suite, by IBM. Websphere MQ provides an implementation of the JMS standard and, of course, supports every pattern that JMS covers.

Additionally, Websphere MQ allows the configuration of one particular endpoint to exclusively receive messages off a channel. Therefore Websphere MQ fully supports space-coupling. Consequently item ‘7’ in Table 3.1 was assigned a ‘+’.

MSMQ

MSMQ is the MOM implementation by Microsoft. It provides MOM support to the BizTalk process application server and to the new Windows Communication Foundation.

MSMQ directly supports blocking-send, blocking-receive, and non-blocking receive. However, like the JMS, it does not directly support non-blocking send. Nevertheless the same work-around to achieve non-blocking send for JMS can be performed using MSMQ. Hence we assign MSMQ a ‘+/-’ for item ‘2’ in Table 3.1.

MSMQ, like most MOM solutions does not support time-coupling. Hence we assigned a ‘-’ to items ‘5’ and ‘6’ in Table 3.1.

MSMQ has full support for space-decoupling over a channel but would require non trivial ‘work-arounds’ to achieve space-coupling. Therefore we assigned MSMQ a ‘-’ for item ‘7’ in Table 3.1.

With respect to space-decoupling over a topic MSMQ offers a peek operation in its API – allowing receiver endpoints to peek at messages in the queue. Using peek to support space-decoupling over a topic (or publish-subscribe) is a work-around, in much the same class as Java spaces. Therefore, it was assigned a ‘+/-’ for item ‘9’ in Table 3.1.

MSMQ, supports multicast directly but to our knowledge does not support multicast/join, or any of the request-response patterns.

MPI

The Message Passing Interface, developed by a consortium of leading IT vendors and select members of the research community, was designed to enable parallel and distributed systems to exchange messages effectively. Using MPI, parallel applications are able to exploit processing on multiple CPUs for example, because the API is extremely efficient in its use of memory and the CPU. MPI fully supports all forms of thread (de-) coupling, providing ex-
plicit operations for blocking-send, non-blocking send, blocking-receive, and non-blocking receive – as is shown in Table 3.1.

MPI does not support time-decoupling – as shown in Table 3.1. MPI does not support space-decoupling over a channel. MPI is able to notify all members of a group with copies of the same message. This behaviour is strongly related to *space-decoupling over a topic*, however these groups are determined during build-time, or design-time. There seems to be limited support for joining a group at runtime, and no support for joining more than one group. We therefore rated MPI a ‘+/−’ for *Space-decoupled (Topic)*.

MPI fully supports multicast, message joining and preprocess acknowledgment.

### 3.6.3 Scenario - Mobile Devices and a BPM System

Consider a hospital that needs to integrate a Business Process Management (BPM) system and a proximity sensor system to send requests to medical staff based on their skills and location. Each medical staff is given a mobile device that relays location information to a central system. The BPM system uses this information to allocate work items to perform patient care services in an efficient, timely manner.

The challenge is to design an integration model accounting for the varying levels/types of connectivity between distributed systems, and mobile resources. Clearly the mobile devices will not always be connected to the central system (due to varying levels of signal availability), and therefore the use of non-blocking send is advisable. Hence messages to and from mobile devices could be stored until the signal is restored. New mobile devices might need to be added to the system and device swapping may occur – which should not break the integration. Therefore space decoupling is required, but we do not want to notify many instances of the same resource with the same work request, thus ruling out publish-subscribe. Finally, it is likely that the mobile devices have intermittent connectivity and therefore time decoupling between mobile devices and the central system is necessary.

Based on these requirements it is clear that one should use a combination of Pattern 2 (Non-blocking-send), Pattern 3/4 (Non-blocking-receive/Blocking-receive), and Pattern 6 (Time-decoupled). Combined this could amount to either configuration ‘C14’ (Non-blocking-send, *Blocking-receive*, Time-decoupled, Space-decoupled-channel) or configuration ‘C16’ (Non-blocking-send, *Non-blocking-receive*, Time-decoupled, Space-decoupled-channel) from Section 3.4.2.

The table (Table 3.1) shows that only JMS, Websphere MQ, and MSMQ could support such a combination of requirements. Furthermore the table
suggests a potential sticking point for each of these solutions as they all only partially support Pattern 2 (Non-blocking-send).

It is worth noting that the choice of coupling configuration has an impact on the performance of the underlying middleware. In particular, Praphamontripong et al. [120, 121] have reported experiments showing relations between queue size and throughput in the context of time-decoupled communication. Meanwhile Saif and Parashar [130] shed light into the performance tradeoffs of non-blocking communication in MPI, showing in particular its sensitivity to the size of system buffers and the size of messages. Finally, Maheshwari and Pang [98] studied performance tradeoffs of space-decoupled communication in message-oriented middleware, showing in particular correlations between number of subscribers and middleware performance.

### 3.7 Prototype

In the spirit of validating the proposal, we designed an API on top of the Java language, namely JCoupling. We introduced [13] as a means of enabling correlation of messages to instances of a business process through the use of properties, channels and filters. In this chapter, we do not consider the issue of message correlation and filtering, focusing instead on the coupling integration patterns and the request-response patterns. JCoupling combines the ideas presented in this chapter with those presented in our previous work [11].

JCoupling is not a middleware per se: It does not provide application services traditionally associated with middleware such as reliable delivery (i.e. retries), transactions, security, etc. Its purpose is merely to illustrate how the proposed concepts can be used to support different types of communication through a unified API. The source code of the prototype is available from [www.sourceforge.net/projects/jcoupling](http://www.sourceforge.net/projects/jcoupling).

A JCoupling endpoint may change its coupling with remote endpoints differently for each interaction, if the programmer chooses this. Thus it is possible, to send a sequence of messages from the one endpoint, and for each interaction adopt a completely different interaction pattern. JCoupling is written in Java 5.

Figure 3.22 presents a summary of the JCoupling API. It can be seen from this figure that the abstract class `Integrator` is central to the JCoupling API. Concrete implementations of `Integrator` perform the transport responsibilities, required by any `Sender` and `Receiver`. For instance, a possible implementation of `Integrator` could enable interactions over TCP sockets. The `JCouplingFactory` interface provides a dynamic means of creating
instances of alternative implementations of the Integrator class. For instance an implementation of this interface could create either a JMS, TCP, or SOAP/HTTP Integrator implementation based on the contents of a text-based configuration file.

As shown in Figure 3.22, the Integrator interface has two primary methods:

- **receiveRequest()** immediately returns a RequestKey identifying the interaction request, and then places the request onto the JCoupling server (which is not shown). When a message is ready, on the JCoupling server, a call-back is made by the server onto the requesting Integrator correlating the original request using the same key. This notifies the Receiver to retrieve the message off the server. The blocking or non-blocking interaction styles are implemented within the Receiver, which is akin to the way they were modelled in the CPNs of previous sections.

- **sendRequest()** operates in a similar manner, except that a MessageID is used instead of a request key, and that the client is a Sender.

Non blocking methods on both the Sender and the Receiver immediately return java.util.concurrent.Future objects. An object of type Future is
essentially a handle to obtain a desired object once it is ready. The methods of Sender and Receiver that return lists are concerned with multicast and aggregate message receipt.

### 3.7.1 Example 1: Hello World

The following listings demonstrate the implementation of an interaction using the Blocking-send, Blocking-receive, Time-decoupled, and Space-decoupled (Channel) patterns, as presented in Section 3.3. Such a coupling configuration mimics the common behaviour observable in most Message Oriented Middleware.

**Listing. 1.** Performs a time-decoupled, space-decoupled, blocking-send.

```java
1     IntegratorFactory factory = new LocalIntegratorFactory();
2     try {
3         Integrator integrator = factory.createIntegrator();
4         Sender sender = new Sender(integrator);
5         Channel channel = (Channel) integrator.lookup(CHANNEL_URI);
6         Message message = new Message();
7         message.setContent("Hello World");
8         sender.blockingSend(message, channel, TimeCoupling.decoupled);
9     } catch (JCouplingException e) { ... }
```

In Listing 1, line 2 creates a factory capable of instantiating Integrator objects, which basically allows client code to abstract away from the underlying transport protocol. Lines 5–6 create the sender and obtain a reference to the channel. Line 11 sends the message over this channel in a time-decoupled manner. Line 12 is needed because lines 4 and 11 can throw either a TransportException, NotFoundException, or a PermissionException (all sub-types of JCouplingException).

**Listing. 2.** Performs a space-decoupled, blocking receive.

```java
1       IntegratorFactory factory = new LocalIntegratorFactory();
2       try {
3           Integrator integrator = factory.createIntegrator();
4           Receiver receiver = new Receiver(integrator);
5           Channel channel = (Channel) integrator.lookup(CHANNEL_URI);
6           Message message = receiver.blockingReceive(channel, Receiver.NEVER_TIMEOUT);
7           // At this point, the message has been received
8           ... 
9       } catch (JCouplingException e) { ... }
10      } catch (TimeoutException e) { ... }
```

91
In Listing 2, lines 3–5 create the receiver, and obtain a reference to the channel. Line 7 performs the receive - returning a `Message` object. The output from executing Listings 1 and 2 is: Received msg ID: M-9223372036854775808&C1365919705802591056 Contains: Hello World

### 3.7.2 Example 2: “Scatter-gather”

A “scatter-gather” interaction is akin to an RPC Broadcast: given a collection of endpoints, a request is sent to each of these endpoints, and a response is later gathered from each of them. For demonstration purposes we will portray a purchase order scenario. When a purchase order is received by `Hardware-R-Us` an inventory check reveals that certain line items are understocked. So `Hardware-R-Us` performs a scatter-gather request on three wholesalers for a quote, with the best quote being pursued.

Hohpe and Woolf [77] propose an implementation of the `scatter-gather` pattern using JMS. The endpoint playing the role of `Hardware-R-Us` sends a request onto a JMS topic, and each wholesaler receives the request, creates a quote, parses the request for a return address, and posts the quote on the queue. The “`Hardware-R-Us`” endpoint receives each quote, one by one, keeping the best.

To implement the same scatter-gather using JCoupling requires less coding. `Hardware-R-Us` publishes a non-blocking-send, as shown in Listing 3.

**Listing 3.** Publishes a search request to all libraries, and filters for the best responses.

```java
try {
    IntegratorFactory factory = new LocalIntegratorFactory();
    Integrator integrator = factory.createIntegrator();
    Sender sender = new Sender(integrator);
    Topic topic = (Topic) integrator.lookup(TOPIC_URI);
    Message message = new Message();
    message.setContent(BOOK_REQUEST);
    Future<ResponseContainer> futureResponses =
    sender.nonBlockingSend(message, topic, TimeCoupling.coupled);

    ...// do something else while responses are coming back

    ResponseContainer responseContainer = futureResponses.get();
    List<Response> subscriberResponses = responseContainer.getResponses();
    List<Response> goodResponses = filterResponses(subscriberResponses);
}
```

The wholesaler endpoints don’t need to explicitly receive messages, in-
spect return addresses, and send replies, they only need to implement an interface and add it to a Receiver object (Listings 4 and 5).

Listing. 4. The message processor interface.

```java
public interface MessageProcessor<V> {
    public V processMessage(Message message) throws Exception;
    public <U extends Serializable> U getResponse();
}
```

Implementations of MessageProcessor should provide the application logic needed to process the message, and to format a response. Method processMessage gets called first, then getResponse. The result of invoking getResponse gets returned to the sender.

Listing. 5. Creates a search receive/response server for a wholesaler.

```java
try {
    IntegratorFactory factory = new LocalIntegratorFactory();
    Integrator integrator = factory.createIntegrator();
    Receiver receiver = new Receiver(integrator);
    Topic topic = (Topic) integrator.lookup(TOPIC_URI);
    receiver.subscribe(topic, TOPIC_PASSWORD);
    QuoteRequestProcessor quoteRequestProcessor = new QuoteRequestProcessor();
    receiver.nonBlockingReceiveAsService(topic, quoteRequestProcessor);
} catch (JCouplingException e) { ... }
```

In Listing 5, lines 4 - 6 create the receiver, obtain the topic reference, and subscribe the receiver to the topic. Lines 8 - 9 instantiate an instance of the MessageProcessor interface and register it with the receiver. When a message arrives, the method processMessage of QuoteRequestProcessor will be invoked (similar to onMessage in JMS).

In JCoupling each type of destination (i.e. address, channel, or topic) are capable of queueing their incoming messages. Consequently we can guarantee that the order in which messages arrive on the bus is the order in which they are consumed. Therefore, if the sender and receiver on one destination are blocking, JCoupling can guarantee preservation of message sequence. Another feature of JCoupling is its ability to perform multicast, and multiple message joining.

In building this prototype we now know that it is not overly difficult to build a coupling platform that natively incorporates the best features of Message-Oriented Middleware and RPC-based middleware, and that an implementation of these proposals does not need to compromise on the strengths of either middleware family. Furthermore we have demonstrated that the API to support this wide range of interaction styles can be relatively simple and concise.
3.8 Related Work

Cypher and Leu [48] provided a formal semantics of blocking/non-blocking send/receive which is strongly related to our work. Their primitives were defined in a formal manner and related to the MPI [134]. This work does not consider space decoupling. Our research differs by combining thread-decoupling with the principles of time and space-decoupling (originating from Linda [64]). Furthermore, our work unified these dimensions to define coupling integration patterns that can be used as a basis for middleware comparison.

Charron-Bost, Mattern, and Tel [40] provide a formalisation of the notions of synchronous, asynchronous, and causally ordered communication. This study introduces a notion of generalisation among these forms of communication according to sequences of messages at the global perspective and cyclic dependencies between them. They propose an increasing gradation of strictness starting with asynchronous computation (akin to all forms of time-decoupled communication), through FIFO computations (akin to message sequence preservation), through causally ordered computations, and finally to the most strict form - synchronous computations (akin to thread-coupled, time-coupled communication). Their formalisation of “asynchronous” is akin to our notion of thread-decoupling; nevertheless they do not distinguish between time-decoupling and thread-decoupling as alternative forms of asynchronous communication. This highlights the fact that their work focusses on formal classifications of distributed message sequences, while our is more architectural in nature. Their work does not address concepts such as space-decoupling, and only mentions request-response very briefly.

Cross and Schmidt [46] discussed a pattern for standardising quality of service control for long-lived, distributed real-time and embedded applications. This was a proposal for “configuration tools that assist system builders in selecting compatible sets of infrastructure components that implement required services”. In the context of that paper no proposals or solutions were made for this, however the proposals of this chapter perhaps provide a fundamental basis for the selection of compatible sets of infrastructure.

Thompson [143] described a technique for selecting middleware based on its communication characteristics. Primary criteria include blocking versus non-blocking transfer. In this work several categories of middleware are distinguished, including conversational, request-reply, messaging, and publish-subscribe. The work, while insightful and relevant, does not attempt to provide a precise definition of the identified categories and fails to recognise subtle differences with respect to non-blocking communication. Our work, on the other hand contains precise definitions for blocking and non-blocking
communication and it addresses the dimensions of time and space.

Schantz and Schmidt [131] described four classes of middleware: Host infrastructure middleware (e.g. sockets), Distribution middleware (e.g. CORBA [109], and RMI [138]), Common Middleware Services (e.g. CORBA and EJB), and Domain Specific Middleware Services (e.g. EDI\(^{14}\) and SWIFT\(^{15}\)). This classification provides a compelling high-level view on the space of available middleware. Their classification focuses on architectural concerns and domain-oriented criteria. While this classification does offer important differentiations between alternative forms of middleware it does not highlight these alternatives in terms of the coupling architectures available for each middleware alternative. Consequently our work can be construed to complement theirs.

Tanenbaum and Van Steen [141] described the key principles of distributed systems. Detailed issues were discussed such as (un-)marshalling, platform heterogeneity, and security. The work was grounded in selected middleware implementations including RPC, CORBA, and the World Wide Web. Our work is far more focused on coupling at the architectural level, and therefore complements the technical issues discussed by Tanenbaum and Van Steen.

### 3.9 Conclusion

This chapter has presented a set of formally defined notational elements to capture architectural requirements with respect to coupling. The proposed notational elements are derived from an analysis of middleware in terms of three orthogonal dimensions: (1) threading, (2) time and (3) space. The patterns proposed in this work identify subtle differences between time-decoupling and thread-decoupling. Either time-decoupling or thread-decoupling, alone, can in fact provide what is commonly regarded as asynchronous behaviour, somewhat overloading the term’s meaning; and yet in some middleware examples they are both present (e.g. MPI, JMS) – underlining their distinctness.

The selection of middleware, in practice, is unlikely to be solely concerned with coupling semantics. Criteria gaining greater, and perhaps deserved at-

---


\(^{15}\)SWIFT: Society for Worldwide Interbank Financial Telecommunication. SWIFT provide a value added network enabling messages concerning transactions to be exchanged between banks of the world ([http://www.swift.com](http://www.swift.com) accessed Jan 2007).
tention in middleware selection include reliability, standards, compliance, maturity, and *du jour* techniques. This work does not propose to replace or invalidate those criteria but rather promises to provide a means of quick comparison of various middleware alternatives with respect to coupling semantics.

In this chapter, we have argued that the terms ‘synchronous’ and ‘asynchronous’ are too imprecise to constitute a foundation for defining models of integration. This inspired us to define a set of coupling integration patterns that conceptualize the notions of synchronous and asynchronous communication, among others, in an unambiguous manner. We also linked multicast/join patterns, and six request-response patterns into the core concepts. This set of patterns unifies and organises existing knowledge in the domain of integration coupling. We have also presented an embodiment of the proposed concepts in the form of an API, namely JCoupling.

Our key motivation for this chapter was to address the lack of an overarching framework that can unambiguously express architectural requirements with respect to (de-)coupling. We believe that the proposed coupling patterns introduced throughout this chapter offer a sufficient solution to this goal, and that the CPN models underpinning these patterns offer an unambiguous expression of their semantics.

In collaboration with Gecko ([http://www.gecko.de/](http://www.gecko.de/) accessed December 2009), we are currently building a commercial form of the JCoupling API. We plan to add a middleware aggregation layer to the implementation, allowing JCoupling to pass interaction requests over alternative forms of *outward facing* middleware, including JMS and WSDL-based implementations. A meta-data layer will allow us to capture the semantics, programatically, of the *outward facing* middleware quite concisely, so that applications can “reason” about the semantics of different forms of middleware. As part of this work we intend to conduct load testing. We also reported in [13], some extensions to the JCoupling API to deal with message filters, and we have shown how this enables the implementation of sophisticated forms of message correlation in the context of business process management systems. We are currently extending this work to integrate it into a workflow engine, namely YAWL [152].

**Disclaimer**  The assessments we made of middleware products and standards with respect to the coupling integration patterns are based on the tool or standard’s documentation and, in some cases, experimentation. They are true and correct to the best of our knowledge.
Chapter 4

Matching Communicating Processes

At present, the definition of business processes that interact with one another in a distributed environment is hampered by a number of factors. Firstly, these process to process interactions need support from some integration infrastructure. This integration infrastructure could take the form of MOM, an ESB, or a SOAP-client/server etc. These infrastructures are not equal, and the choice of infrastructure will permit or prevent certain form of collaboration from taking place [11, 27, 55, 48]. There exist conceptual problems with state-of-the-art business process definition languages in regards to process-to-process communication abstractions. To receive message batches for example, BPEL implementations force iterations over receive tasks – which are susceptible to errors and other issues. Business Process Management systems rarely support message filtering, based on business rules. Indeed, the distinction between process abstractions and communication abstractions is blurred in existing approaches.

One of the main problems with state of the art SOA-based BPM solutions is their strong coupling to state-of-the-art integration technologies (e.g. WS-BPEL’s dependance on SOAP). Nevertheless, in practice much, and perhaps most, B2B interactions continue to be performed using mature, entrenched technologies such as Email (with attachments) and FTP etc.

This chapter motivates and defines a match-making service within a process-integration architecture. A match-making service is an enabling technology for matching any incoming messages to a particular running instance of a process. This is absolutely essential where many instances of the same process share one communication channel. This proposed match-making

\[1\]

\[\text{For instance JMS cannot filter messages based on their body content.}\]
service sits between the process execution layer and any communication middleware. It simplifies the definition of interactions between distributed processes. The proposed abstractions have been implemented on top of a communication API, namely JCoupling [12], that abstracts away from the underlying middleware and protocols. Using this implementation, we have conducted preliminary experiments to compare this approach with respect to the approach embodied in the BPEL [18].

The next section presents motivating scenarios and requirements for process-to-process communication. Section 4.2 introduces a communication model for distributed business processes addressing these requirements. Section 4.3 discusses the implementation and experimental evaluation of a prototype that implements these abstractions. Section 4.4 discusses related work and Section 4.5 concludes the chapter.

4.1 Requirements Underpinning a Match-Making Service

Traditional requirements for integrating distributed systems typically include tightly-coupled and/or loosely coupled architectures [55], guaranteed delivery, encryption etc. Integrating processes introduces significant new complications. In this section we distill some of these specialised requirements. These requirements are drawn from two studies on patterns in the area of integration of enterprise applications: the Enterprise Integration Patterns by Hohpe and Woolf [77], and the Correlation Patterns by Barros et. al. [25].

The correlation patterns described by Barros et al. review conversation and message consumption patterns for distributed business processes. Selected patterns led to requirements motivating our proposals. From Hohpe and Woolf we categorised the patterns into (1) Patterns forming requirements for a BPM messaging solution, (2) Patterns that are supported by most middleware, (3) Patterns composable from any data-aware, message-aware system, (4) Patterns concerning deployment and administration. The communication abstractions proposed in this paper focus on the first category. Following is our categorisations of Hohpe and Woolf’s patterns.

**Patterns that constitute requirements for a BPM messaging solution**

Aggregator, Channel Purger, Competing Consumers, Correlation Identifier, Data Type Channel, Event Driven Consumer, Message Expiration, Message Filter, Polling Consumer, Publish-Subscribe Channel, Remote Procedure Invocation, Request-Reply, Return Address, Scatter-Gather, Selective Consumer, Transactional Client
Well understood patterns supported by any middleware solution
Command Message, Document Message, Event Message, Guaranteed
delivery, Message Endpoint, Message, Messaging, Point to Point
Channel, Message Channel

Patterns composable from any data aware, message aware system
Canonical Data Model, Claim Check, Composed Message Processor,
Channel Adapter, Content Based Router, Content Enricher, Content
Filter, Detour, Dynamic Router, Envelope Wrapper, Format Indicator,
Idempotent Receiver, Invalid Message Channel, Message Broker,
Message Bus, Message Dispatcher, Message History, Message Router,
Message Sequence, Message Store, Message Translator, Messaging
Bridge, Messaging Gateway, Messaging Mapper, Normalizer, Pipes
and Filters, Process Manager, Recipient List, Resequencer, Routing
Slip, Service Activator, Shared Database, Smart Proxy, Splitter, Test
Message, Wire Tap

Patterns concerning deployment/administration Control Bus, File
transfer

In the remainder of this section, we discuss the different requirements.
For each requirement we cite previously-published related patterns.

4.1.1 Conversations
Support for conversations is necessary when business processes need to ex-
change more than one related message, and in particular when the processes
are stateful, or execute over long periods. Furthermore typical process de-
ployments, require many instances to share common channels and conversa-
tions enable messages finding their way to the correct instance. SOA propos-
als such as BPEL may create the illusion that there is one process instance,
when in fact there are many – because all process instances are hidden be-
hind the one SOAP endpoint. We know of five approaches to implementing
conversations through correlation:

Property-based Correlation assumes that many process instances share
the same channel. Messages get routed to the appropriate instance by
applying process-defined functions to incoming messages and then by
matching the results to process instance values.

Business Object-based Correlation possibly works in a similar way to
property-based correlation, however in this case the process-defined
functions being applied to incoming messages are generated by the process container. The process models express the notions of conversation through much higher abstractions, for instance business objects on the process. The correlation-set construct in WS-BPEL is a good example of this style of correlation.

**Token-based Correlation** again involves many instances sharing the same channel. In this case mandatory “correlation tokens” inside the header of each message identify to which process instance they should be routed.

**Instance Channels** achieve correlation by using separate channels for each instance. The REST [57] architectural style advocates such an approach, claiming that this approach more closely follows the architecture of the Web. The REST approach proposes to apply the same principles into distributed application development. Some of its advantages include greater transparency, scalability, and perhaps flexibility. Zur Muehlen et al. [156] describe the application of the REST architecture to BPM achieving conversations through instance channels. Such an approach seems intuitive to many process designers.

**Protocol-based Correlation** – In most forms of request-response the correlation of the request with the response is provided by the technology, for free. For example HTTP request-response, or CORBA-RPC.

WS-BPEL supports business object-based correlation and protocol-based correlation. WS-BPEL engines may support token-based correlation through the use of implementation-specific tokens for instance routing, however these tokens being implementation-specific may not have general application.

*Related to:* Key Based Correlation, Property Based Correlation, Reference Based Correlation, Conversation Overlap, Hierarchical Conversation, Initiator, Follower [25], Correlation Identifier [77]

*Scenario 1: Purchase Order* A purchase order is received. The process sends separate queries to different suppliers for each line item. Each response is correlated over the purchase order identifier, and over the line item number. This example demonstrates the need for nested conversations.

### 4.1.2 Property-Based Message Selection

Property-based message selection helps a process pick the best message off a channel. This significantly reduces the complexity logic within the process
designed to iterate through an internal array of messages. There should be simpler abstractions for this.

*Related to: Message Filter, Selective Consumer [77].*

**Scenario 2: Line Items** A parts buyer wants to proceed with the best quote.

### 4.1.3 Atomic Multiple Source Consumption

When messages need to be joined, from more than one source, atomic multiple source consumption greatly reduces complexity in the process model. This is because the messages from each source may need to match over a certain field or property, and packaging them together removes the need to do this in the process. Furthermore it is much simpler to design exception handling if there is only one point of failure in the process. To demonstrate this, consider the possible outcomes of consuming them separately: (1) both message arrived, or (2) one failed to arrive, or (3) both failed to arrive, or (4) they both arrived however there was a property mismatch, or (5) there were two processes and each consumed only one message. Hence, atomic multiple source consumption would any exception handling code, by allowing it to be linked to only one communication task.

*Related to: Atomic Consumption [25].*

**Scenario 3: Purchase processing** Task “ship-goods” will not begin until there is a confirmation of credit from the accounts department, and all line items have been notified as being “in stock”.

### 4.1.4 Aggregated Consumption

Consuming messages in one “pass” of a process loop greatly reduces complexity. There is no need to iterate through a set of receive actions, or to encode loop stop-conditions, which can often be a little arbitrary if the intent is to consume all available messages. Furthermore it may be necessary to choose messages only if their properties, taken collectively, satisfy certain criteria.

*Related to: Aggregator [77].*

**Scenario 4: Shipping Company** When at least 100 items have arrived destined for the same district, and a truck is available, the truck gets dispatched.

### 4.1.5 Aggregated Consumption involving Time

Business processes are in many cases very time sensitive, for example the hours of business (9am to 5pm, Monday to Friday, EST). Thus there exists the need to include the notion of time into the process layer, for example
by allowing message selection based on temporal constraints. Temporal constraints can typically be either absolute or relative i.e. “15 – 20 November, 2006”, or “within the last 7 days”. Both styles are necessary with the latter being more challenging due to the group of eligible messages being in a continual state of flux with the passage of time.

Related to: Time-Based Correlation, Moving Time Window Correlation [25]  
Scenario 5: Time If, over the last five working days, more than five percent of the incoming messages that arrive at the department contain complaints then an emergency quality control process gets launched.

4.1.6 Contention

Contention, or competition, for the same resources, is a natural phenomenon. For instance, auctions and goods tendering rely on contention between competitors. Messages, unlike auction items, have little or no intrinsic value and therefore contention over messages may not seem compelling. However, contention over messages is an enabling technique for load balancing of message consumption; wherein many instances of the same process share the workload by only consuming messages/events when they are ready.

Related to: Competing Consumers [77].

In BPM systems supporting the deferred choice pattern [7] there are two layers of contention: (a) contention for the same message between different task instances (b) contention between tasks of the same process instance for shared control flow triggers.

Scenario 6: Competing Processes Copies of a process are distributed onto different hosts. The first process instance to claim the message provides the service, thus distributing the processing workload.

4.1.7 Handling Events

Processes, by their very nature, need to be able to handle events that they do not solicit. Actors in the environment being under their own spheres of influence, generate events that the process may not anticipate, but the process will need to react to in a given way. Consider the following example:

Related to: Event Driven Consumer [77].

Scenario 7: Event Handling A travel booking may be canceled, should the customer decide to do so. The process needs to be able to undo certain actions at points in the process if such an event occurs, however the process never explicitly “awaits” the receipt of a cancelation event.
4.1.8 Channel Passing

The ability for a process to “learn” about a new actor during execution by virtue of a reference to a new channel being contained in a received message. Then, the consequent ability for the process to communicate with this new actor.

Related to: Return Address [77].

Scenario 8: Channel passing A corporation begins travel booking through an agency. The agency then supplies the corporation with a channel where the status of the booking can be queried as it progresses.

4.1.9 Garbage Collection

When messages arriving on a channel are too carefully filtered there becomes the strong possibility of having a large number of unconsumed messages on each channel. Messages have expiry dates however expiry dates cannot be updated onto a message once it is buffered on the channel. Special garbage collection filters could be added and removed from the message layer dynamically during process execution.

Related to: Message Expiration, Channel Purger [77].

Scenario 9: Garbage Collection Once a process completes all unconsumed messages addressed to this process instance are removed from the set of input channels.

4.1.10 Interaction Cancelation

The ability to cancel incomplete interactions.

Related to: Transactional Client [77], Cancelation [7].

Scenario 10: Cancelation A supplier having posted a receive purchase order request, discovers that the warehouse needs to be replenished first, and consequently cancels its earlier receive-request.

4.1.11 Summary

These requirements, being drawn from the respective patterns studies, ought to be supported by most state of the art solutions, however we found that this was not the case. For instance, despite WS-BPEL being possibly the most widely accepted standard in this domain, it only provides support for the first requirement. It is one thing to support a wide range of problems in communicating business processes, however the language constructs exposed to the creators of these processes need to be suitable to their purpose, and
they should be as intuitive, or conceptually aligned to their purpose as is possible.

4.2 Communication Model for Business Processes

In this section we introduce a model supporting interactions between processes addressing the requirements outlined above. In the proposed model, instances of a process model are hosted in a process container. The container is aware of a set of channels that are referred to by the process model. These channels may be combinations of email addresses, JMS Queues, SOAP Endpoints e.t.c. Channels can be used by a process container to send messages (outbound channel), receive messages (inbound channel), or perhaps both directions (bi-directional channel).

Outbound channels are referred to by a unique name, and a scheme (e.g. email, JMS) and possibly a type restriction and/or a description of an endpoint descriptor (the latter may be determined only at runtime). The proposed model abstracts away from the specific language used to describe message types. A particular embodiment of the concept of channel is one where message types are captured in XML Schema (or WSDL). Individual messages sent through the channel are then validated against its XML schema (or WSDL message type definition). Similarly, we abstract away from the mechanism used to describe destination endpoints. If using HTTP as a transport protocol for example, an endpoint can be described as a URL, while if using SOAP/HTTP, it can be described as a WSDL operation binding. Outbound channels support a range of message sending primitives which are described in detail elsewhere. In the rest of the paper however, we focus on inbound channels.

Inbound channels have the same components as outbound ones but they additionally have a set of properties. A property is a function that takes as input a message and produces a literal value. This is similar to the concept of property alias defined in BPEL. However, as shown later, the scope of applicability of properties in our model is wider than that of BPEL. In BPEL, property aliases (and their composition in the form of correlation sets) are only used to correlate pairs of outbound and inbound messages. In contrast, in our model, properties can be used to perform other forms of message selection and aggregation.

A relation (i.e. a database table) is created for each inbound channel used in any process model. Each relation contains two predefined attributes: one
of type \textit{message identifier} and the other of type \textit{timestamp}. Additionally, the relation contains one attribute (column) per property associated to the channel.

Properties are used to define filters. A filter is a function that is evaluated against the set of messages available for consumption over one or multiple channels. When the evaluation of a filter returns a non-empty set of messages, we say that the filter \textit{matches} these messages. Filters can fulfill two purposes: (i) \textit{garbage-collecting filters} are registered by a business process to discard unwanted or unnecessary messages over inbound channels; (ii) \textit{message consumption filters} are used to consume one or multiple messages from one or multiple channels. Orthogonally, filters may be \textit{one-off} or \textit{persistent}. A one-off filter is immediately withdrawn after it has matched a message or set of messages, while persistent filters are preserved until explicitly withdrawn.

A filter is represented as a query over the relations(s) associated with the channel(s) it refers to. These queries are always constrained to produce a relation wherein each tuple contains \textit{message identifier} attributes.

When a message arrives onto a channel, a tuple is inserted into the relation associated with that channel. This tuple always contains \textit{message identifier} and \textit{timestamp} attribute values, as well as attribute values obtained by applying each of the channel’s properties to the incoming message. The inserted tuple represents the incoming message for the purpose of evaluating message filters. After being abstracted as a tuple, the incoming message is either:

- Immediately routed to a message receipt action if one has registered a filter that matches the message (possibly in combination with other messages).
- Discarded if the message matches any of the garbage-collecting filters registered for that channel.
- Queued until it matches a garbage-collecting or message consumption filter.

Conceptually, a filter is re-evaluated every time that a new message arrives to any of the channels it refers to (or continuously in the case of filters whose query depends on the current time). In practice however, the evaluation can be made incrementally and only when required.

Primitives for registering and withdrawing filters are provided as part of the communication framework. Registration and withdrawal of filters can be initiated either by the process container or by individual process instances.
When a message consumption filter is registered, the filter is run once against the set of messages available in the channels referenced by the filter. If the filter matches one or several messages, these are removed atomically from their channel(s) and given back to the process container or process instance that registered the filter. If the filter is one-off, it is withdrawn. Should no match between a filter and the existing set of messages be found upon registration of the filter, the filter is maintained and re-evaluated whenever required as explained above, until the filter is either explicitly withdrawn or it matches a message (or set of messages). Once a match is found, the message(s) are routed to the corresponding process container or instance and the filter is removed.

Garbage-collecting filters work similarly: when registered, the filter is evaluated against the contents of the channels targeted by the filter. If a match is found, the matched message(s) are discarded. If the filter is one-off, it is withdrawn otherwise, it is preserved and it is re-evaluated when required.

Figure 4.1: Petri net capturing the treatment of inbound messages.

The proposal is formally captured as a Coloured Petri net in Fig. 4.1. The net shows how inbound messages are stored and matched against filters. A token in place “incoming message” represents a message received by the communication layer. A transition called “put” moves tokens from this place to a place called “message buffer”. This latter place holds a single token containing a list of all unmatched messages over all channels. Two of the places “garbage collecting filters” and “message consumption filters” are meant to contain one token per active filter. Transition “collect garbage” fires when there is a garbage-collecting filter that matches at least one of the messages in the message buffer. This transition puts back a modified message buffer in which all messages matching the garbage-collecting filter have been removed. Similarly, transition “match” fires if there is a consumption filter matching at least one message in the buffer. This transition also puts back a modified
message buffer in which the matched messages are removed. In addition, it produces a tuple containing the filter and the set of matched messages into output place “matched messages”. These tokens can then be routed to the process container or process instance that registered the filter in question. The latter is identified by a request identifier (“reqID”). For simplicity, the net only captures the case of “persistent filters”, but it is easy to extend the net to deal with one-off filters: the only difference being that such filters should not be put back by transitions “collect garbage” and “match”.

The proposed communication model abstracts away from the way channels and filters relate to business process activities or events. This way, the model can be integrated into a wide range of process definition languages. In BPEL, for example, inbound communication actions appear in two forms: as a standalone receive activity type and as the second leg of activities of type invoke, where the first leg corresponds to an outbound communication action. Thus, BPEL can be extended with the proposed communication primitives by enabling receive and invoke activities to refer to channels and filters as defined above. Channels can then be linked to partner links and operations in BPEL.

Similarly, the proposed model can be used to extend the YAWL process definition language to support a richer set of communication patterns. For example, we can define a type of message receipt task in YAWL such that: (i) upon enablement, the task registers a one-off message consumption filter defined as part of a task decomposition; (ii) the task then waits until the filter returns a match; (iii) should the task be cancelled before a match is found, the filter is withdrawn. Also, we can allow message consumption filters to be attached to the initial condition of a YAWL process model, to capture scenarios such as: “a new process instance should be started whenever a given type of message (or combination of messages) has been received.”

4.3 Implementation and Evaluation

The implementation builds on a middleware service, and API, called JCoupling (available from www.sourceforge.net/projects/jcoupling). JCoupling supports a superset of the communication styles supported by mainstream communication middleware. It also abstracts away from transport protocol details, thus allowing us to concentrate on the core aspects of our proposal. It supports uni- and bi-directional communication, time, space, and synchronisation decoupling, and provides support for fault propagation. JCoupling can operate over open-JMS\(^2\) and its openly extensible architecture

would allow the creation of additional adapters.

Figure 4.2 presents an architectural diagram of the prototype. It shows properties and filters being used during a simple interaction between two process tasks. In Fig. 4.2: (1) A message is sent by task “T1” over channel “Ch1” (denoted by arrows labelled “1” going from the engine to the controller). (2) Properties “P1” and “P2” are used to extract values from the message. Next the message is put onto JCoupling (3.a), and a new tuple (row) is added to the relation (table) for channel “Ch1” (3.b). (4) The receiver task “T2” posts a filter over channel “Ch1”. (5) Using the filter, the controller performs a query, over the relation for channel “Ch1”. That query produces a tuple and the matching message is extracted from JCoupling (6). The callback to “T2” contains the message (7).

Figure 4.2: Architecture of the proposal.

This section gives further details on the implementation of properties, interactions and filters, and reports on an experimental evaluation of the prototype.

4.3.1 Implementing Properties

The prototype implementation contains an interface called Property. This interface and three implementing classes are presented in Fig. 4.3.\(^3\) In-

\(^3\)The proposal is not limited to these three property classes. For example: an EdiProperty class could be created for use with EDI messages.
stances of Property extract various scalar values from messages. Parametric classes (i.e. "generics") are used to define the return type for the method accessValue().

We envision a configuration tool for creating property instances such that process designers do not need to write Java code. For example if the process designer wishes to create an XPath property, to extract "PurchaseOrderID's" from messages then it would only be necessary to supply a property name, a channel binding, an XPath expression, and a Type (i.e. Text, Numeric, or Timestamp⁴) defining what type the XPath expression produces (e.g. ‘PurchaseOrderId’, ‘P0_Chann’, ‘/order/@po-id/text()’, ‘Text’).

Here is a listing of the Property interface, in Java 5.

Listing. 6. Property Interface.

```
1...2 public interface Property<T> { 
3 /**
4 * The name of the property.
5 * @return the name.*/
6 public String getName();
7 
8 /**
9 * Retrieves the value of the property from the message.
10 * @return a type 'T' value of the property.*/
11 public T accessValue(Message message) throws Exception;
12 
13 /**
14 * @return the relational attribute type for storing the property value.*/
15 public Type getDBColumnType();
16 
17 /** The relational attribute types currently supported by property. */
18 public enum Type {Text, Numeric, Timestamp}
19 }
```

⁴It can be seen that in our implementation Properties can produce values of either Text, Timestamp, or Numeric. This list could possibly be extended in the future.
The prototype currently contains three example implementations of the interface Property:

**TimestampProperty** inspects the message for the time it arrived on the channel. The method accessValue(Message message) uses the generic typing in Java to return a ‘Date’. The method getDBColumnType() returns the enumerated type Property.Type.Timestamp.

**XPathProperty** extracts values out of the XML body content of the message using XPath statements passed in through its constructor. The method accessValue(Message message) returns an object that is either a ‘Date’, a ‘Double’, or a ‘String’ – depending on the Property.Type and the XPath statement passed into its constructor.

**HeaderProperty** extracts the values of any named properties from the message header. The method accessValue(Message message), uses Java’s generics to return a ‘String’. The constructor receives the name of the message header property, which is used to name its corresponding relational attribute for that property, and is used to extract the value from the header.

In our implementation each channel has its own, dedicated database relation for storing its own property values that we refer to as the property relation. Adding new properties to a channel results in new columns being added to that channel’s property relation. For example adding property PurchaseOrderId to channel PO-Chan causes the attribute PurchaseOrderId to be added to its property relation (PO-Chan). Any newly added attribute is declared to allow ‘null’ values, and hence new columns may be added during execution without causing an SQL error, if there is data in the relation. To match messages SQL queries may be executed over the property tables to find messages that match certain property values. These may be in many cases quite simple. On the other hand the SQL expressions can be quite sophisticated, as we shall see in Sect. 4.3.3.  

With the arrival of an incoming message the match-making prototype adds a tuple to the *property relation* for that message’s channel using a PostgreSQL database. As mentioned in Sect. 4.2, each channel’s property relation has two default attributes (columns): messageid and timestamp. Table 4.1 presents a relation for channel QuotesCh.

Table 4.1: Relation corresponding to channel QuotesCh.

<table>
<thead>
<tr>
<th>messageid</th>
<th>timestamp</th>
<th>quote</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0000M0001</td>
<td>2006-11-15 16:35:50</td>
<td>4000</td>
<td>1100</td>
</tr>
<tr>
<td>C1111M0002</td>
<td>2006-11-16 18:12:20</td>
<td>3000</td>
<td>1000</td>
</tr>
<tr>
<td>C2222M0001</td>
<td>2006-11-16 20:42:53</td>
<td>2500</td>
<td>1001</td>
</tr>
<tr>
<td>C3333M0003</td>
<td>2006-11-17 16:57:21</td>
<td>2000</td>
<td>999</td>
</tr>
</tbody>
</table>

4.3.2 Resolving Contention

A process may have to wait days before an appropriate message can be received. Hence, the prototype stores send/receive requests and performs callbacks when the interactions are complete. Indeed requests to send/receive, have their own lifecycle, including interaction cancellation (i.e. when process state change makes unnecessary an incomplete interaction). A race condition between two receive tasks is a perfect example of this. For example, a task “receive-bill-payment” and a task “receive-purchase-order-cancelation” operate such that the firing (beginning) of task one disables the other.

![Timeout Receive Payment Msg](image)

Figure 4.4: Example of a YAWL process with a deferred choice involving a receive task.

Figure 4.4 illustrates a race condition between a task “Timeout” and a task “Receive Payment Msg”. Here, the token in the place enables both tasks. The task that fires first will claim the token, un-enabling the other. Any action that the un-enabled task has started (for example placing a receive request with the match-maker) will need to be cancelled. This is equivalent to a “pick” activity in BPEL, or a “deferred choice” in the Workflow Patterns [7]. Accordingly the prototype (being a control layer between a process and the message layer), exposes a ‘withdraw’/’cancel’, primitive for send and receive interactions – also implementing the cancellation requirement (see Section 4.1).

Another form of contention occurs where two tasks both want the same message/s. Contention may be a requirement (for example load balancing, Scenario 6), or it may be accidental, or perhaps even unavoidable due to the
nature of the business process. Regardless contention, being a natural and sometimes necessary phenomenon, mandates a graceful handling it.\(^\text{6}\)

The following algorithm is proposed to address any contention in the match making service.

**Listing. 7.** Handling contention between many processes and many messages.

```plaintext
1 function resolveRequest(requestID, filter)
2     VAR matchedMessages
3     VAR messageBuffer //Unconsumed messages in storage
4     WHILE active(requestID, filter)
5         IF matches(messageBuffer, filter) THEN
6             VAR matchedMessages = match(filter, messageBuffer)
7                 VAR requestLocked = lockRequest(requestID)
8                     IF requestLocked THEN
9                         VAR allMessagesLocked = true
10                            FOR each message in matchedMessages
11                              IF message is unlocked
12                                  VAR success = lockMessage(message)
13                                      IF not success THEN
14                                          allMessagesLocked = false
15                                      END IF
16                              END LOOP
17                     END IF
18                     IF allMessagesLocked THEN
19                         FOR each message in matchedMessages
20                             removePropertyTuple(message)
21                         END LOOP
22                         removeMatching(filter, messageBuffer)
23                         deactivate(requestID, filter)
24                         END IF
25                     END IF
26                     ELSE
27                         //Request cannot be locked. Perhaps task is
28                         //cancelling request.
29                     END IF
30     END LOOP
31     RETURN matchedMessages
32 END function
```

Listing 7 provides the algorithm that was implemented in the prototype to handle contention between many process instances and many incoming

\(^{6}\)This is a distinguishing point from WS-BPEL, which throws runtime exceptions when contention between two receive-tasks occurs. Also, WS-BPEL’s greedy routing of messages to process instances precludes the possibility to support contention.
messages. The listing is an extension of the semantics presented in Fig. 4.1. Once the request and filter produce a non empty set of matching messages (Line 8), the request is locked to ensure that the receive task cannot withdraw the request (Line 13). Every matching message is checked to ensure that it has not been already locked by another process (Lines 16 – 24). If all messages were successfully locked, then for each message, that property tuple is deleted (Line 28), and each corresponding message gets removed from the buffer (Line 31). The request and filter are withdrawn (Line 32) and the matching message/s are ready to be sent to the waiting receive task (Line 41). If the request cannot be locked (the process may be canceling it – Line 13) the attempt is abandoned and the algorithm will exit because the request is inactive (Line 6). Should any of the matched messages already be locked (Lines 21 – 23) then the request and filter are unlocked (Line 37), and the filter is rescheduled (Line 6). The first process to successfully lock its message/s succeeds.

4.3.3 Implementing Filters

This section shows how the extensions to JCoupling outlined above, supports the motivating requirements of Sect. 4.1. Relations over sets of properties declared on a channel are able to abstract from message content. These relations enable the use of targeted SQL queries to produce results containing message identifiers. The only restriction we place over the query expressions is that their outer projections must only be over attributes of the domain `messageid`.

A single property, or combinations of properties can be used to select messages. These queries can be generated by a workflow engine that understands business rules for message consumption expressed over properties and channels. In Chapter 6 we present these abstract business rules as an extension to a workflow language. Alternatively, for those situations requiring sophisticated aggregate operations, or complex joining expressions, the process designer may write their own SQL over the abstract properties.

**Property Based Selection** Scenario 2 captured the need to select and proceed with the best quote, which is an example of selecting messages based on property values. The query in Listing 8 uses two properties defined over the channel “QuotesCh” (see Table 4.1). These are “price” and “quantity”. When a receive request containing a query is invoked over the prototype, it will apply the query – returning messages in a callback to the requestor when results are found.
Listing. 8. This query combines ‘price’ and ‘quantity’ to find the best value offer.

1  SELECT messageid
2  FROM QuotesCh
3  WHERE quantity >= 1000
4  AND price/quantity =
5  ( SELECT min (CostPerUnit)
6    FROM ( SELECT price/amount AS CostPerUnit
7      FROM QuotesCh ) AS Q1 )

Conversations Scenario 1 outlined the need to correlate messages to an outer conversation for “purchase-order-ID” and an inner (nested) conversation for “line-item-id”. Messages will only be correlated to a nested conversation if they satisfy correlation filters for the inner conversation, and all parent conversations. To achieve this we append an AND-Clause to the query. The following query (see Scenario 1) extracts runtime data from two process variables visible to the receive task: namely PurchaseOrderID and LineItemID.

Listing. 9. Achieving nested correlation through querying correlation properties.

1  SELECT messageid
2  FROM PoResponseCh
3  WHERE PoID = $PurchaseOrderID$
4  AND ItemID = $LineItemID$

A process integration language could easily generate such queries for conversation from a “conversation” construct. This construct would declare which message properties are to be used for correlation, and whether each communication task involved initialises the conversation, or follows it. Furthermore if any task involved in the conversation wants to apply message filters we may have the task append more AND clauses to the generated query. No proposal that we know of offers this expressive power or flexibility.

Atomic Multiple Source, and Aggregate Consumption Achieving a combination of atomic multiple source consumption and aggregate consumption is possible by applying a query to two or more property relations. For example Scenario 4 required aggregate consumption of 100 packages destined for the same area, and a truck availability message from another channel. Listing 10 extracts runtime data from a process level variable called deliveryDistrict. The query either returns at least 100 tuples, or returns nothing.
Listing. 10. This query produces a relation of messageid’s wherein the attribute
(Pack.messageid) refers to messages from Channel Packages, and the attribute
Truck.messageid refers to a message on Channel TruckWaiting. Related mes-
sages are linked, thus removing the need to relate messages in the process.

```
1 SELECT Pack.messageid, Truck.messageid
2 FROM Packages As Pack, (SELECT messageid FROM TruckWaiting LIMIT 1) AS Truck
3 WHERE Pack.deliveryDistrict = '$deliveryDistrict$
4 AND 100 <= ( SELECT count(*)
5 FROM Packages
6 WHERE deliveryDistrict = '$deliveryDistrict$
7 )
```

Aggregated Selection Involving Time Scenario 5 sought to find if at least 5%
of messages that are less than 7 days old contain complaints; drawing
on solutions to both time and aggregated selection. Additionally complaints
were sought on all channels. A join between property relations will not work
as every tuple from each relation represents one discrete event that needs to
be considered separately. So unlike Listing 10, the result relation will have
one attribute. A view over the union of source channels (property relations),
solves this. In Listing 11 this view is named “Merged”. It combines properties
(messageid, timestamp, and complaint) from each of the source property
relations.

Listing. 11. Using union and view to combine input sources for aggregate
operations.

```
1 CREATE VIEW Merged AS ( 
2 SELECT messageid, timestamp, complaint 
3 FROM CustomerCh 
4 UNION SELECT messageid, timestamp, complaint 
5 FROM PartnerCh ) 
```

Using the above view, aggregate calculations over the messages, taken
collectively, becomes feasible. Listing 12 demonstrates this. Lines 5 – 12
produce false unless 5%, or more, of the messages are complaints. Lines
3, 8, & 12 show the use of relative time expressions over the timestamp
property. In cases like this we imagine that the process creator would write
Listings 11 and 12 manually.

Listing. 12. This query, adapted from a continuous query in [22], produces a non-
empty result of messageid’s when 5%, or more, of last week’s messages contain
complaints.

```
1 SELECT messageid
2 FROM Merged
3 WHERE timestamp > (CURRENT_TIMESTAMP - INTERVAL '7 days')
```

115
AND complaint = true
5  AND ( SELECT count(*)
6    FROM Merged
7    WHERE complaint = true
8    AND timestamp > (CURRENT_TIMESTAMP - INTERVAL '7 days')
9  ) >= (
10  SELECT 0.05 * count(*)
11  FROM Merged
12  WHERE timestamp > (CURRENT_TIMESTAMP - INTERVAL '7 days')
13 )

4.3.4 Garbage Collection

So far we have introduced the notion of a filter that is addable to the proto-
type that helps in receiving messages. We extended the API slightly to have
a special type of filter that removes stale messages. Like receive filters the
garbage collection filters contain a set of channels, and an SQL statement
designed to be applied over the property tables for each channel. The rules of
execution of receive filters is exactly the as that of garbage collection filters,
except that garbage collection filters do not attempt to lock or callback any
receivers. They just remove messages from the channels when a match is
found.

4.3.5 Performance Evaluation

To compare our correlation approach with that of BPEL, we conducted ex-
periments where up to ten thousand interactions were executed over our
prototype and over a BPEL simulator. Each experiment involved creating,
in random order, at fixed intervals, a set of XML messages, all identical
except for the value of one element which was mapped to a property. Re-
ceiving processes were spawned in the same way, each of which waited for
one of the created messages. The code for the experiments is released with
the JCoupling distribution.

The BPEL correlation simulator was built using the same middleware as
our prototype. This simulator receives messages off a designated channel.
Each time a message arrives, an XPath expression is evaluated against it
to extract a property value (i.e. a BPEL propertyAlias). The extracted
property value is then stored in a hash table together with the corresponding
message identifier.7 Concurrently, the simulator handles requests to con-
sume incoming messages based on property values. When the simulator
finds a match between a receive request and a message, the corresponding

---

7In the interest of fairness, each entry is written to disk after being added to the
in-memory hash table, as our prototype stores property values in persistent tables.
entry is deleted from the hash table, symbolising that the message has been correlated.

<table>
<thead>
<tr>
<th>Number of Interactions</th>
<th>50</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1000</th>
<th>2500</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms)</td>
<td>Proposed Approach</td>
<td>611</td>
<td>951</td>
<td>2230</td>
<td>5064</td>
<td>9003</td>
<td>25276</td>
<td>71409</td>
</tr>
<tr>
<td></td>
<td>WS-BPEL Approach</td>
<td>524</td>
<td>938</td>
<td>2103</td>
<td>4046</td>
<td>7825</td>
<td>20506</td>
<td>44204</td>
</tr>
<tr>
<td>Performance Difference</td>
<td>14%</td>
<td>1%</td>
<td>6%</td>
<td>26%</td>
<td>46%</td>
<td>19%</td>
<td>38%</td>
<td>43%</td>
</tr>
</tbody>
</table>

Table 4.2: Results of performance tests.

The test results are presented in Table 4.2. The table shows that our approach slightly underperforms that of BPEL for small numbers of messages. This difference grows for larger numbers of messages. The accentuated difference can be explained by the fact that in the implementation of our approach, queries to match uncorrelated messages with pending receive requests are run against a persistent database, whereas in the BPEL simulator, the corresponding lookup is done in-memory\(^8\). For larger numbers of messages, this leads to a performance penalty due to database cache management. A more consistent performance could be achieved by using simple Java prepared statements for queries with the same structure but different runtime search values. Indeed, it is not necessary to make the correlation data structures persistent, only the messages themselves need to be persistent.

The performance penalty of our approach should be weighed against the additional functionality that it brings in. Indeed, our approach supports aggregate messaging, multi-source consumption and message contention. Moreover, there are opportunities to optimise the brute-force approach used in our implementation through incremental query evaluation.

### 4.4 Related Work

Communication in the context of distributed business processes has traditionally been researched from the perspective of protocol or contract definition. For example, the CrossFlow system [68] enables process designers to define contracts governing the communication between multiple workflows, possibly distributed across organisational boundaries. These contracts can be statically checked for consistency. Similarly, van der Aalst and Weske propose a method for capturing inter-workflow communication protocols and detecting deadlocks that can arise when inter-connecting processes with incompatible communication protocols [8]. This body of work is complementary to our proposal, as we do not deal with static analysis, but rather with the routing of messages to processes at runtime.

---

\(^8\)Although entries are written to disk, lookups over the hash table are done in-memory.
BPEL exhibits strong support for conversations with the exception of *Instance Channels*, which are not supported because WSDL *ports* are not generated during process execution. BPEL supports event handling and provides *some* support for channel passing. BPEL does not support the selection of messages based on their properties despite the use of properties in correlation sets. Nor does it support atomic-batched consumption. Contention over messages is not supported as messages are eagerly consumed off channels and allocated to process instances immediately. Time-based message selection is not supported either. A BPEL process consumes everything directed to it, thus it is not possible to filter unwanted messages where the scenario demands handling large datasets. WS-CDL \[90\] has some distinguishing features, with respect to BPEL, such as abstractions for channel passing. However, in terms of the motivating requirements outlined in this paper, WS-CDL has very similar strengths and weaknesses to BPEL.

Widom et al. \[22, 110\] propose an approach to optimising the evaluation of continuous queries over one or many data streams. They address some of the problems associated with scalability of such queries and propose incremental evaluation techniques based on the type of query. We plan to apply some of their findings to enhance and optimise the evaluation of filters used in our proposal. In particular, Olston et al. detail techniques to make continuous queries more scalable while slightly sacrificing accuracy in some cases \[110\].

### 4.5 Conclusion

This chapter proposed a match making service, tying instance of business processes with communication middleware. Powerful filtering techniques replace “spaghetti” workflow approaches in message intensive scenarios and non-trivial long-standing interactions. The key concepts of the implementation are powerful and relatively simple – namely channels, properties, property relations and filters. *Channels* abstract from messaging pathways (e.g. topics, queues, email address). *Properties* abstract from the way to retrieve unique values for correlation, irrespective of message structure. *Property relations* provide the foundation for property filters. *Filters* are a means of expressing business rules for message consumption based on simple queries. These enable all forms of correlation, message selection, aggregated message consumption, and time based message consumption, over a single or multiple channels. The possibility of contention between process/task instances is overcome by locking messages, and filter requests. The proposal has been implemented on top of a communication API, namely JCoupling, that abstracts away from the underlying middleware and communication protocols.
Chapter 5

Patterns of Process Integration

5.1 Introduction

Business process integration, like BPM, lacks an accepted formal foundation. Unlike database technology, which has a formal foundation, business process technology tends to differ widely from offering to offering, hindering its integration potential. Indeed a formal foundation is extremely useful [7, 6].

With no precisely defined overarching framework to guide development of a technology it follows its own path. This results in waves of technologies countering the perceived mistakes of previous technologies [75]. These waves can be observed in the field of distributed systems integration. It has been actively researched and developed for nearly 40 years. In spite of this observers have noted that it is subject to wave after wave of new integration technologies (see Section 1.1.2). By contrast database systems have had a formally defined, fit for purpose conceptual framework. Database platforms have been stable enough, for long enough, that they are highly standardised around ODBC and SQL.

In the absence of a generally accepted formal framework for defining all of the static and dynamic aspects of process integration a set of patterns helps express what a solution should be capable of supporting.

This chapter presents requirements that appear to be essential for modelling process integration. Previously workflow patterns have been used as a comparison instrument for workflow and BPM platforms [7, 25]. The Workflow Patterns provided a measure by which we can assess the modelling power of any workflow language, and they formed requirements during the creation of YAWL. Likewise the patterns in this chapter are intended to help us understand and assess languages for process integration. This set of patterns is compiled from analyses in earlier chapters, related work [55] as well as related...
integration patterns compilations [26, 25, 77] and key features of integration technologies [134, 85, 104].

5.2 Coupling Dimensions

The coupling dimensions patterns are drawn from earlier parts of this thesis. Each is briefly revisited here with some higher level discussion about their application. Formal semantics for these patterns can be found in Chapter 3.

The first of the three dimensions expresses thread-coupling. Thread-coupling influences process design quite heavily. It encapsulates whether the process threads will be waiting or not for the presence of communication. The second dimension, time, is related to whether two participants need to both be participating in an interaction at the exact same moment for the interaction to occur. The last dimension, space, deals with whether the processes interacting with one-another address each other directly, or using aliases, and whether a message sent is duplicated to all listeners or sent to only one. We will apply these dimensions to uni-directional interactions, although they can be generalised (see Section 3.5 for discussion).

5.2.1 The Thread Coupling Dimension

Thread-decoupling enables “non-blocking communication”. Non-blocking communication allows the sender to fire off a message and then forget it. It allows the receiver to not wait for messages, but essentially be woken up when one has arrived. Thread-coupling, by contrast allows a sender to get live feedback about the success or failure of the interaction. If a process tries to contact another process about something timely, or important, it may be helpful to get immediate, or timely, feedback about the interaction. This is a lot easier to do using a time-coupled approach. Next we will see that thread coupling and decoupling can be broken down into four basic patterns.

Pattern Coup1: Blocking Send. A blocking send is the basic pattern describing when the sending process waits while the message is being sent. A blocking send is characterized by a task in a process that waits until the message has left the process execution environment.\(^1\) In a blocking send situation the sending process will know when the message has successfully left the process. Once we know the message has left; the process will be able

\(^1\)If the interaction was bi-directional it would block longer (Section 5.4).
to continue, comfortable in this knowledge. Refer back to Figure 3.1(a) for illustration.

*Akin To:* Synchronisation coupling [55].

*Supported By:* MPI (Blocking Send) [134], JMS (send) [71], CORBA (sendMessage) [109].

*Example:* An approval to purchase over $100,000 worth of company stocks needs to be sent. The trader needs to ensure that the message is enroute before a covered call can be written.

**Pattern Coup2: Non-blocking Send.** A *non-blocking send* is the alternative to blocking send. It describes the pattern where the sending process is able to continue immediately, regardless of whether the message has left or not. A non-blocking send means that the task of transmitting the message is decoupled from the overall flow of the process.

Support for the non-blocking send pattern is fairly uncommon in middleware. Websphere MQ and Microsoft Message Queues (MSMQ) claim to support it, however their support is only partial. They typically support it by hosting a message queuing server on the sender’s computer. Non-blocking send is uncommon in RPC-based technologies but is used in highly parallel technologies such as MPI [134]. Refer back to Figure 3.2(a) for illustration.

*Akin To:* Synchronisation decoupling [55].

*Supported By:* MPI (Non-blocking Send) [134].

*Example:* A child care agency worker performs a field assessment of a group of children at one residence. The e-form submissions are stored on the tablet PC until it docked into its station.

**Pattern Coup3: Blocking Receive.** A *blocking receive* pattern occurs when a process wishing to receive a message waits for the message to arrive while halting the flow of the business process. The process resumes, only once the message has arrived. This means that the flow of the process is highly coupled to the message arriving.
If the fact of a message not arriving is as important (and needs to be handled) as the message arriving then it may be wise to consider making the process wait for the message to arrive. Refer back to Figure 3.3(a) for illustration.

_Akin To:_ Synchronisation coupling [55].

_Supported By:_ MPI (Blocking Receive) [134], JMS (receive) [71].

_Example: 1_ A retailer is waiting for a customer to pay before shipping the goods.

_Example: 2_ The city courier truck must not pick up international shipments until the cargo has cleared customs.

**Pattern Coup4: Non-blocking Receive.** A non-blocking receive pattern occurs when the receiving process is not waiting for the message to arrive. In many situations we do not care if a message fails to arrive. However, what if something needs to be done _if_ a message arrives. An incoming message affects the process, but its absence is not going to impact the processes success or failure. In other words, if something comes up we handle it; otherwise the process proceeds _business-as-usual_. A non-blocking-receive pattern is the right pattern to use in this situation. It is illustrated in Figure 3.4(a).

Non-blocking receive is a suitable pattern to capture the process scenario where an external event may be triggered. This pattern forms the foundation of most asynchronous architectures. However, non-blocking, or asynchronous techniques, when applied to scenarios where our process depends on the arrival of a message force us to check, and recheck whether the message has arrived yet, and if it does not arrive in time we have to take additional action. These scenarios are more easily modelled using blocking-receive techniques, where a timeout can be used to take remedial action if the message does not arrive.

_Akin To:_ Synchronisation decoupling [55].

_Supported By:_ MPI (Non-blocking Receive) [134], JMS [71].
Example: 1 A microchip warehouse receives a request to supply 400 4 GB RAM cards.

Example: 2 A message is received by a performance car manufacturer that the customer wishes to change the car colour.

A non-blocking send is a necessary condition, but not a sufficient condition to achieve total thread-decoupling, which is to say that the receive action must also be non-blocking. Rephrased, if both the send and receive are blocking (non-blocking) then a total thread coupling (decoupling) occurs. A partial thread decoupling occurs when the send is blocking and the receive non-blocking, or vice-versa.

5.2.2 The Time Dimension

Time, or timing, is a fundamental aspect of an interaction. Like the thread-engagement dimension, the time dimension has a large impact on the amount of feedback (or lack thereof) possible between two integrated processes. In an interaction time is either coupled or decoupled.

Pattern Coup5: Time-Coupled. Time-coupled interactions require that both processes are participating in an interaction at the same moment. In time-coupled arrangements both processes begin the interaction at the same time, and complete it at the same time. A depiction of time-coupled interactions is presented in Figure 3.5(a).

Because the sender and receiver are involved in a time-coupled interaction concurrently, it is possible to know the success/failure immediately. When thread-coupling is used alongside time-coupling it is even easier to know success/failure. Middleware such as RMI, RPC, and SOAP/HTTP make good use of this fact.

Payment gateways frequently use time-coupling to give feedback to the customer. Wouldn’t it be a poor service if you entered your credit card details online, only to get an email the next day letting you know that the bank systems were unavailable at the time of payment? With time-coupling, however, there is a trade-off between ease-of-feedback and just getting on with the job. With time-coupling both processes have to be online and available. If any link in the chain is broken then integration cannot occur.

Akin To: Time coupling [55].
Supported By: CORBA [109], MPI [134].

Example: In an insurance company, each day 20 000 letters need to be automatically generated. These print jobs are sent to selected network printers. If any printer happens to be out of toner/paper then the automation needs to select another printer and redirect that batch of print jobs.

**Pattern Coup6: Time-Decoupled.** Time-decoupled interactions, by converse, don’t support feedback about the success/failure of the interaction, however the popularity of time-decoupled integration technologies such as JMS, email, SMS is testament to the fact that feedback doesn’t always matter. Refer back to Figure 3.6(a) for illustration.

If a process wants to send a message and then get on with other things then this pattern is optimal.

The pattern basically assumes that the messages are stored enroute; and implementations are built this way. Middleware solutions such as Websphere MQ and MSMQ are examples.

Akin To: Time decoupling [55].

Supported By: CORBA [109], JMS [71].

Example: 1 A workforce of 2 000 contractor workers in a mining corporation needs to be notified via email about timesheets and rostering approvals on a weekly basis.

Example: 2 A weather station is regularly sending its readings to the weather bureau’s central systems.

Example: 3 Commercial banks reconciling account transfers overnight.

5.2.3 The Space Dimension

The dimension of space encapsulates where, or to whom, the message is being sent. We have identified three patterns of integration residing in the space dimension. All are extremely common, and have their own unique strengths.

**Pattern Coup7: Space Coupled.** A space-coupled interaction is essen-
tially one where the sending process knows about the receiving process. It is also known as direct addressing and is common in many integration techniques and technologies. Refer back to Figure 3.7 for illustration.

Space-coupling is simple and direct. There are no proxies and layers of indirection that the true destination of a message is hiding behind. Hence it is observable in many of the clean and simple integration techniques available (e.g. these include SOAP/HTTP, fax, email, FTP etc.).

**Akin To:** Space coupling [55].

**Supported By:** CORBA [109], MPI [134].

**Example:** A cosmetics manufacturer receives a large purchase order from an exporter of makeup and hair colouring products. It is received over fax.

**Pattern Coup8: Space-Decoupled.** A space-decoupled interaction (akin to indirect addressing) occurs where the sender process interacts with the receiver process through an intermediate address. Such a technique is slightly more elaborate than direct addressing. The sender only knows about the indirect address, and the receiver registers itself with the indirect address as the receiver for it. Hence the two processes are decoupled in space, i.e. from each other in the addressing domain. Refer back to Figure 3.8(a) for illustration.

Typically the more sophisticated middleware technologies offer space-decoupling. These include, for example, Java Message Service (JMS), WebSphere MQ, and CORBA/IDL. Uses of space-decoupling include clustering and load balancing applications (because many receivers can share the workload of incoming messages). Additionally, space-decoupling enables hot-swapping of an old process with a new one. This is achieved by de-registering the old process to receive messages while registering the new one in its place.

**Akin To:** Point-to-Point Channel [77], Space decoupling [55].

**Supported By:** CORBA [109], JMS [71].

**Example:** An airline booking confirmation process listens for confirmations on a JMS queue. A mainframe application listening on the JMS queue is end-of-life’d and replaced with a BPEL process. This change is invisible for
the travel agents using the service.

**Pattern Coup9:** **Space-Decoupled Topic (Publish-Subscribe).**

Space-decoupling can take on a slightly different flavour called space-decoupled topic (Publish Subscribe). Space-decoupling, as presented, allows a message to be passed, indirectly from the sender to a receiver, however, publish-subscribe allows a message to be passed from a sending process, with the difference being that every registered receiver gets a copy of the sender’s message. This pattern is related to the observer pattern [62] used in software engineering. In the observer pattern an object known as the subject maintains a list of its observers, and notifies them of any state changes to the state of the subject. Refer back to Figure 3.8(b) for illustration.

Integration technologies such as MSMQ, Websphere MQ, and JMS support this pattern of interaction.

*Akin To:* Publish-Subscribe Channel [77], Space decoupling [55].

*Supported By:* JMS (Topic) [71].

*Example: 1* A process that is configured to broadcast the fluctuating trade price of a stock to various day trading applications.

*Example: 2* A government Department of Accommodation periodically renews property maintenance contracts by sending requests for tender to a government contracts notice-board. Local businesses must supply a registered business number in order to receive and submit proposals over the notice-board portal.

In summary: space-coupling allows any sender to direct the message to one known receiver. Space-decoupling allows the sender to interact with any receiver process that happens to be registered with the indirect channel at that time. Note that there can be many such receiving processes sharing the channel, in which case we have load-balancing or clustering - since the interaction will only reach one of the receivers. Space-decoupling (topic), on the other hand, allows the sender to interact with all receiver processes that happen to be registered with the indirect channel at that time. Each one will get its own copy of the message.
5.3 Message Consumption Patterns

Messages are intuitively understood to be removed from their source channel when the application receives them. In Chapter 4 we examined how a service can filter for messages over a channel, but when a match was found the match-making service always removed the message off the channel. There are scenarios where the message is not always removed this way and these are examined in this section.

**Pattern Cons10: Take.** In most cases when a message gets received by a process task it gets removed from the channel. The take pattern captures this concept.

![Figure 5.1: Message take.](image)

*Supported By:* Java Spaces (Take) [85], MSMQ (Receive) [104].

*Example:* A purchase order is received by a warehouse over email. A business process engine consumes the email and launches an instance.

**Pattern Cons11: Copy.** Certain scenarios require a message to be left on a channel and just copied down to the process. In order to support this there needs to be a way of specifying the message consumption model. This capability is unusual in most forms of middleware. MSMQ\(^2\) and Java Spaces\(^3\) are exceptions in that they support it. The copy pattern can be implemented using other techniques if needed, but this is prone to error and is not trivial.

*Supported By:* JavaSpaces (Read) [85], MSMQ (Peek) [104].

\(^2\)MSMQ has a `peek()` operation that copies a message to the client.

\(^3\)Java spaces has a `read()` operation that copies an object to the client [85].
Example: An airline announces a change to a flight time to a travel agent. Changes to flight times will influence many customers’ bookings so the message must be shared amongst each booking process.

**Pattern Cons12: Clean.** Messages can clutter communication channels for three reasons: (1) all process tasks refuse to consume the message because it doesn’t satisfy a business rule, (2) the process consuming messages off a channel is only copying them, not consuming them, (3) the data in messages needs to be timely or it is useless. i.e. it has an expiry date. In any case there needs to be a way of cleaning up superfluous messages.

Example: Stock ticker announcements get cleaned up after they become one minute old.

### 5.4 Request-Response Patterns

Many integrated processes need immediate feedback, in response to the messages they send. Bi-directional interactions are an enabler of this feedback.
A request-response is the archetypal bi-directional interaction. The term bi-directional interaction is preferred because request-response implies that the interaction is both time-coupled and thread-coupled.

As mentioned earlier, one-way interactions that are both time-coupled, and thread-coupled dramatically increase the certainty of the sender about the success or failure of the interaction. However, in such one-way interactions there is no way to immediately tell (1) whether the receiving process understood the message, (2) what the receiver has done with the message. So if the aim is to increase certainty of state the best approach is to use bi-directional interaction styles.

This is fairly intuitive but it is important to know that if we choose time-decoupled and thread-decoupled techniques like email, fax, or FTP we will not easily be able to have the same certainty. And of course building an aggregation of email interactions to mimic a request-response is non-trivial. The alternative types of integration technologies on offer provide varying degrees of certainty about the remote state of a process integration. Certainty about remote process state is difficult with fire-and-forget techniques. Conversely, time-coupled, two-way technologies (such as CORBA) offer strong certainty about remote process state. Message-Oriented Middleware technologies (such as JMS), contain rich APIs allowing alternative interaction styles. These can offer extreme uncertainty about the interaction or total certainty depending on the interaction style chosen.

Bi-directional messaging adds additional possibilities, for instance delivery receipt and the reporting of receiver-side faults. A delivery receipt (i.e. system acknowledgement) is an event returned to the requestor, that its message has been successfully received. A delivery receipt (as distinguished from a response) does not imply that the targeted endpoint has processed the message – just that it has received it. Finally, a receiver side fault being propagated back to the requestor indicates that an error occurred during the processing of the request message.

**Pattern Resp13: Acknowledgement.** a signal gets returned to the requestor indicating that the message was successfully processed. This is useful in any application where we need to be certain that the message has been processed. It is supported by JMS.

**Supported By:** JMS (Acknowledge) [71].

---

4Middleware technologies such as CORBA and SOAP/HTTP support request-response.
Example: An automated reconnaissance drone is sent additional instructions while aloft. The automated process that sent the instructions will alert stakeholders if the drone does not acknowledge.

**Pattern Resp14: Response.** This pattern describes a standard response containing meaningful information. This is useful in any application where the requestor needs information back from the responder.

![Message Acknowledgement](image)

Figure 5.4: Message acknowledgement.

Akin To: Remote Procedure Invocation, Request-Reply [77].

Supported By: CORBA [109].

Example: A biofuel retailer is replenishing bio-diesel stocks. A local process automatically consults three local suppliers of bio-diesel, finding the best deal for five thousand litres.

**Pattern Resp15: Throw Fault.** The throw fault pattern is needed when a problem occurs in the responder. Rather than just hide the fault message inside the standard response the fault is packaged in a purpose-built container and sent back. This sort of pattern is easily attainable without effort provided that the middleware supports fault throwing by the responder. Many situations where the requestor wishes to know how well the request
was received are observable and thus it is frequently supported in RPC-based middleware.

Figure 5.6: Throw fault.

_Akin To:_ Event Message [77].

_Supported By:_ CORBA [109], RMI (RemoteException) [138]

_Example:_ A form is being rendered and several database queries need to be used to inject data into the form. If the database queries fail on the server, the client needs to load a plain, unpopulated form in order to keep the customer engaged.

**Pattern Resp16: Receive response: blocking.** The concept is that the requestor waits (blocks) for the response. This is useful in applications where the response comes back quickly or the process cannot proceed without the response. It is supported by JMS for example.

Figure 5.7: Blocking for response.

_Akin To:_ Remote Procedure Invocation, Request-Reply [77].

_Supported By:_ CORBA [109], JMS (QueueRequestor) [71]
Example: An online bookstore supports users entering their credit card details during checkout. A process consults a credit-card payment gateway with the card details and makes the user wait until the payment gateway provides a response from the credit provider.

Pattern Resp17: Receive response: non-blocking. The requestor thread uses a non-blocking technique to receive the response. This is useful in applications where the response is likely to come back days, or maybe weeks later. It is usually performed by composing two interactions together, one in each direction. Even though this is a little clumsy it can be done with virtually any integration technology.

![Figure 5.8: Not blocking for response.](image)

Supported By: CORBA [109].

Example: A telecommunications company has a workflow system and a job management system. The workflow system sends a request to the job management system and then continues doing other things. The job gets performed by a skilled tradesman who reports back to the job management system. The job management system then sends a response back (3 days later) to the workflow system. The workflow system closes the request.

In the business world we need to be pragmatic about our choice of integration middleware. This is necessary, because we cannot force our trading partners to adopt our integration middleware. If the trading partner uses email only, then email is what we use. But that shouldn’t stop us from designing certainty into our process models. It just forces us, as designers of these solutions, to raise our game. The ideas presented thus far describe the various styles of communication commonly performed by different forms
5.5 Composed Interactions

As mentioned already, correlation, (i.e. the ability to detect two or more related messages from a much larger set of messages, as being related to one another), is a fundamental problem in business processes. Conversations, on the other hand, refer to a moment in time where we have observed two or more end-points exchanging related messages for a given purpose. Note the distinction. Correlation is an ability, whereas a conversation is an actual event that can be observed between many endpoints.

When hundreds, or perhaps, thousands of process instances are each having their own conversations with remote processes, we as designers, need to grapple with the fact that there is a real risk of conversations getting their wires crossed; or more accurately where messages leak between different conversations.

Naturally, a process integration language needs to be able to model both correlation and conversations. There are many middleware technologies and techniques that already solve this problem. Sometimes it is enough just to leverage them. However, sometimes that is not enough and we need to model something more sophisticated. For instance we need something more when using middleware technologies that are not capable of supporting correlation; or when modelling ‘chatty’ conversations involving many end-points. This section presents options of modelling correlations and conversations in processes, starting from techniques that leverage correlation capable middleware, through to highly abstract modelling techniques that leverage sophisticated features of process engines.

5.5.1 Correlation Patterns

**Pattern Corr18: Instance Channels.** Instance channels allow each process instance to be given its own channel. This solves the problem of correlation by effectively distinguishing sets of related messages at their source – by sending related messages over discrete pathways. If a channel is dedicated to one process instance, this removes the possibility of many processes getting their wires crossed. The Répresentational State Transfer (REST) [57] architectural style advocates such an approach, claiming that it more closely follows the architecture of the Web. Basically REST applies successful Web
architectures to service-oriented architectures. Naturally, these same ideas potentially blend with process modelling, given the opportunity. Such an approach seems intuitive to many designers and has influenced proposals around BPM [156, 118].

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{instance_channels.png}
\caption{Instance channels.}
\end{figure}

\textit{Akin To:} REST-oriented Integration [156].

\textit{Example:} A telecommunications company is running fifteen workflow servers. These are geographically separated to ensure constant availability of at least ten servers. In order to support integration, each process instance is given its own URL. The head of the URL points to one of the fifteen servers. The tail identifies a process instance. Any long-standing interaction begun by a process instance contains the URL in the message. External applications use that supplied URL to communicate with the target instance.

\textbf{Pattern Corr19: Request-response Correlation.} Request-response correlation describes the simple scenario where a request and response are correlated by leveraging a middleware technology. The technology does the correlation work, significantly reducing the amount of modeling effort. Like \textit{Instance Channels}, this pattern doesn’t require any additional effort to achieve the aims of correlation, merely taking advantage of circumstances. The request is naturally correlated with the response (i.e. Remote Procedure Calls). These forms of middleware set up a dedicated connection between the requestor and the responder. Once this connection is created the requestor and responder do not need to know about what message is related to what; nor do they need to know about locations or addressing. They only need to know about the connection; even though, in practical terms that connection is short-lived.

\textit{Akin To:} Request-Reply [77].
Figure 5.10: Request response correlation.

Supported By: Oracle BPEL Process Manager [114], Tibco [144].

Example: A warehouse retrieves the price of delivery from a courier’s quotation service.

Pattern Corr20: Token-based Correlation. Token-based correlation uses “correlation tokens” inside messages. As opposed to request-response correlation, token-based correlation allows long-standing, complex interactions to be correlated, with an unlimited number of messages and with an unlimited number of endpoints. However, token-based correlation can require considerable modelling effort inspecting message headers and routing messages.

Figure 5.11: Token-based correlation.

Akin To: Key-based correlation [25].

Supported By: Oracle BPEL Process Manager [114], Tibco [144].

Example: Members of a supply chain are tasked with building computers. They collaborate around supply and distribution of parts required. Messages between supply chain partners need to be correlated to manufacturing batches. This is achieved by inserting a jobID into all messages of the same
5.5.2 Conversation Patterns

While the Correlation Patterns generally leverage opportunity, or features built into middleware, the Conversation Patterns are totally model-driven. The heavy lifting of correlation is done by software according to the modelled intent.

**Pattern Conv21: Property-based Conversations.** This pattern should be modelled by expressing a function (often called a property)\(^5\) over one messaging task, and connecting it with a related function over another messaging task in the same net. This can be done declaratively. The match-making service then applies these functions to incoming and outgoing messages. When messages pass through the match-making service they produce a value. Matching values indicate correlated messages. The properties being applied to messages are part of the process design. This technique is conceptually clean, even if it is a little counter-intuitive at first. The WS-BPEL correlation-set works this way; and many adopters of WS-BPEL struggle with this concept.

![Property-based conversation](image)

**Figure 5.12:** Property-based conversation.

*Akin To:* Property-based correlation [25].

*Supported By:* Oracle BPEL Process Manager [114].

*Example:* A local building society has established a wholesale line of credit with a much larger building society. The preferred data format for collaboration with the larger building society is PDF forms. In order to correlate

---

\(^5\)Properties are basically functions that, when applied to a message, produce a simple string, integer, or date value. They are in practice, often, just an XPath expression.
incoming PDF forms a property is defined on a PDF extractor that is able to retrieve the loan application ID out of the PDF directly. This way the local business analyst only needs to think about conversations regarding a loan application ID. The problem of how the loan application ID is extracted from the PDF is encapsulated from the business analyst.

**Pattern Conv22: Compound Property-Based Conversations.** This is achieved by expressing two or more functions (or properties) over one messaging task, and connecting it with the same number of functions (or properties) over another messaging task in the same net. For example, the task Send Line Item Price Query of an order fulfilment process may apply /PurchaseOrder/LineItem[i] as one property and /PurchaseOrder/@id as the other property. The combination of these values will be used to determine matching messages for the task Receive Line Item Price.

![Figure 5.13: Compound property-based conversation.](image)

**Supported By:** Oracle BPEL Process Manager [114].

**Example:** A retailer needs to send a parts request to a warehouse. The request is sent containing a reference to a purchase order and a line item. The retailer has sent hundreds of requests to other partners containing the same purchase order reference but a different line item reference. Replies need to refer to both fields (PO-ID and Line Item Number) in order to be directed to the right conversation instance.

**Pattern Conv23: Conversation Nesting/Overlap.** Conversations can occur between unlimited endpoints over one business value (e.g. a Purchase Order ID). However many conversations revolve around one common business value and many instances of a repeating business value. In such a model conversation nesting becomes essential (e.g. many line-item conversations nested within the same purchase order conversation).
5.6 Event-based Process Patterns

Intuitively messages and events seem synonymous. However, they are subtly different. A message is something containing data that is sent from one endpoint to another. An event, on the other hand, is an observable occurrence, originating externally, that alters the state of a process. Therefore one type of event is the observable occurrence of a message moving into a process instance. Before the event occurrence the process has no knowledge of the message, and after the event occurrence the message has been assigned to a process variable etc. Other types of events include the passing of a deadline, or an action by a human.

Presented next are a set of event-based process patterns.

Pattern Even24: Event-based Process Instance Creation. Incoming messages can potentially impact processes in two ways: (1) they join onto an existing process and influence it in some way or (2) they cause a new process to be created. Most processes begin with a form, however the problem remains: How do we begin one without a form?

Akin To: Single Event Trigger [51].
Figure 5.15: Event-driven process creation.

Supported By: Oracle BPEL Process Manager [114], Tibco [144].

Example: 1 A shipping company has a service that starts a shipping processes. It is exposed to customers through a SOAP interface.

Example: 2 A JMS message is used to trigger a debt adjustment process in finance.

Pattern Even25: Event-Driven Task. This pattern is useful where a task should only be started if an event occurs. If the event does not occur the task should not be started. Certain tasks are necessary to be done if/when something in the external environment occurs. In the normal flow of business they are not scheduled or waited for as they are not necessary for the safe completion of the process instance. However should something happen outside the control of the process it may be necessary, in the process, to change the original plan and execute an unscheduled sequence of tasks. This sequence is started with an event-driven task.

Figure 5.16: Event-driven task.
Supported By: Oracle BPEL Process Manager [114], Tibco [144].

Example: A bank routinely authorizes personal loans to customers. The forms are filled out and some credit agencies are notified of the application. Normally the bank waits for three days to see if no credit agencies raise any red flags. There is a task called Cancel Application that is only executed if an agency sends a warning about the application.

Pattern Even26: And/Or/XOR Event-Driven Task. In some scenarios a single event is not enough to schedule a sequence of event-driven tasks. A combination of events, however may require a specialised sequence of tasks to handle them. Such a task sequence would begin with the and-event-driven task. A task might be interested in dozens of event types, but it is unknown until runtime which events will arrive. An or-driven-event task would suit such scenarios, and would have similar wait semantics to the or-join. A task may be eager to be triggered by only one of many types of events. This would be supported by an xor-event-driven task.

Akin To: (generalisation of) Multi Event Trigger [51].

![And/Or/XOR event-driven task diagram](image)

Example: 1 A courier company offers a service called Change Delivery Priority while the package is being delivered. This change of priority service maps to a process task that can be fired by either Form Submission or by a message sent from the Parcel Tracking Management System.

Example: 2 A courier company generates a message for each delivery job. Each job message contains a job creation timestamp, a priority, a destination
region, and a job value field. The business rule for despatching a truck is based on the first of one of these three cases (1) there are eighty or more jobs at a loading dock destined for a region, (2) a number of jobs are at risk of deadlining, (3) a number of high priority jobs are waiting at the loading dock.

**Pattern Even27: Event-Driven Choice.** Processes can have XOR splits. Usually the path a process takes is determined checking variables values against business rules. However, an event-driven choice allows alternative process pathways to be determined by events in the enterprise. Put another way, an event-driven choice enables events to determine the directional flow of the process. Sometimes it is necessary to let the business, or an event from the business, make that choice.

![Event-driven choice](image)

Figure 5.18: Event-driven choice.

**Akin To:** (specialisation of) Deferred Choice [7].

**Supported By:** Oracle BPEL Process Manager [114].

**Example:** A new insurance customer takes up a policy. The customer is sent a policy cover-note and a bill. If the bill is paid within thirty days the process updates the policy holder systems. If no payment is received the cover-note is cancelled.

**Pattern Even28: Unsolicited Events.** Not all incoming events are expected, or waited for. There are plenty of situations in the business world where we should only take a certain action if something *out of the ordinary* happens. A process that stops and waits for an event makes that event
solicited. In an unsolicited event pattern the process does not wait for, or solicit, an event. Therefore the non-blocking receive is an enabler of the unsolicited event. An unsolicited event may need to join with or influence the main process flow.

\[\text{Figure 5.19: Unsolicited event.}\]

*Akin To:* Event-Driven Consumer [77].

*Supported By:* Oracle BPEL Process Manager [114], Tibco [144].

*Example:* A customer of a luxury boat manufacturer asks to upgrade the engine configuration during planning/construction.

**Pattern Even29: Event-based Task Interruption.** Tasks can take a long while and seeing them through to completion, if circumstances change, can be wasteful. This pattern solves the problem where an external situation changes while a task is busy. If a task takes minutes, or hours, or even days, and the situation changes making the task no longer necessary it makes sense to be able to interrupt the task.

*Example: 1* A request-response interaction is started but an unexpected fault comes back.

*Example: 2* A travel booking is canceled by the customer.

**Pattern Even30: Timeout.** The timeout pattern describes the concept where a communication task is commenced, and before it is allow to complete
normally it is aborted due to the passage of too much time. A timeout is a specialisation of the Event-based Task Interruption pattern, in which the interrupting event is a time-out.

Supported By: Oracle BPEL Process Manager [114], Tibco [144].

Example: A recruitment agency provides an application cut-off date to a lucrative job offer.

5.7 Filtering

There may be an indiscriminate set of messages on a channel waiting to be consumed. But what if some of those messages are irrelevant to a process? How does one avoid consuming and processing unwanted messages? Message filtering removes the burden on the process of having to check each message in a batch and only keeping the ones we want. It also removes the need to worry about putting the messages back if another process actually is interested in our unwanted messages – because our process wouldn’t consume them in the first place. Message filtering over the contents of the message header is supported by JMS.

Pattern Filt31: Data-Message Comparison Filter. This pattern filters messages based on the value/s of process data and message data. The problem is that the state (data) of the process establishes what the process is interested in. There needs to be a way of using this state to filter out unwanted messages. The solution is to use the process data to select message/s that match its particular case.
Figure 5.21: Message filtering.

Supported By: Oracle BPEL Process Manager [114], Tibco [144].

Example: An auction house only processes bids that are greater than the currently highest bid.

**Pattern Filt32: Time-Schedule Comparison Filter.** Consuming a message based on some business rule expressed in terms of time is sometimes required. There are many instances where a message needs to be selected at a scheduled moment (i.e. end of each business day).

Figure 5.22: Message time filtering.

Supported By: Oracle BPEL Process Manager [114].

Example: A courier company offers a service to any of its customers that allows them to change delivery priority or final delivery address to a package. Any change requests made after 10:00 AM are applied at the beginning of the following business day.

**Pattern Filt33: Time-Timestamp Comparison Filter.** This pattern illustrates how it is valuable to be able to select a message by comparing a
time variable to the “timestamp” of a message. There needs to be a way of consuming a message based on its time of creation. An assumption of this pattern is that messages have timestamps. This is true for most MOM technologies and is frequently enabled by Enterprise Service Buses.

Akin To: Time-interval-based correlation [25].

Supported By: Oracle BPEL Process Manager [114], Tibco [144].

Example: In a share trading scenario a stock ticker is regularly updating stock movements. Only stock updates that are less ten seconds old should influence the share-trading decision-support processes.

5.8 Batch Messaging

Process integration in message intensive scenarios requires handling large message volumes. A process may handle hundreds or perhaps thousands of messages concurrently.

Pattern Batc34: Multicast. In message intensive scenarios it is sometimes necessary to deal with processes outside of the enterprise firewall. It may not be practical to assume that they subscribe to a common topic, or that they use the same middleware technology. This pattern addresses these issues by sending message/s to many receivers in parallel, over their channels. If each message being multicasted is identical it may seem related to the Space-Decoupled-Publish-Subscribe pattern (mentioned previously). However there are some subtle differences. The first one is that multicast requires the sender to know the address of each receiver. Secondly multicast allows the sender to control which endpoints get a message.

Multicast becomes essential where (1) each recipient uses a different type of middleware technology; (2) where the list of receivers are discovered during process execution.\(^6\)

Akin To: Recipient List [77].

\(^{6}\)Dynamic binding of recipient lists could be achieved using a list of channels discovered during process runtime and used to address each interaction.
Example: An insurance company maintains a list of email addresses for its customers. An email is periodically sent to customers based on their notification preferences with the company. It contains special offers and announcements.

**Pattern Batc35: Scatter-Cast.** Message intensive business domains cannot always send the same message to each recipient. Often large corporations need to send en-masse large numbers of customised messages. Scatter-Cast is essential where each message needs to be customised to its receiver.

Akin To: (specialisation of) Splitter [77].

Example: An insurance company periodically sends personalised bills to its customers.

**Pattern Batc36: Message Batch.** Processes occasionally need to handle large numbers of messages. The problem is how to group messages together, in the process data, as a batch? It makes sense to be able to store them as a batch, rather than individually. The batch should be able to support accessing individual messages, indexing etc. The event of receiving many messages would generate the message batch.
Akin To: Aggregator [77].

Example: Two thousand courier requests arrive each hour. These need to be initially processed for logistics planning.

Pattern Batc37: Batch Send. This pattern expresses the ability to send a batch of messages to one endpoint. This solves the problem where the process has a batch of messages in a variable and they all go to the one endpoint/channel.

Example: In a telecommunications company a set of three hundred network status alarms are processed and forwarded to the central alarm management system.

Pattern Batc38: Batch Receive. The complement to multicast is the batch receive. It allows batches of messages to be received in one process action. Sometimes it makes business sense to consume many messages as one unit. It makes for cleaner, more efficient models. This allows the messages to be processed collectively.

Akin To: Aggregator [77].
**Example:** A bank processes thousands of transactions from remote banks, overnight, while reconciling its accounts.

**Pattern Batc39: Minimum Batch Receive.** The concept is to wait until a numeric minimum of messages are available before receiving. The problem is that consuming a lessor number than the minimum will hinder effective processing of the message batch.

*Akin To:* (specialisation of) Message Filter [77].

**Example:** A request for tender cannot proceed until three quotes are ready for appraisal.

**Pattern Batc40: Maximum Batch Receive.** The intent is to set an upper limit to the number of messages that can be received. It can be problematic to handle scenarios where the number of messages being received is too large. Alternatively, there could be an upper limit to how many messages can be handled by a process at a given time (e.g. throttling scenarios). There needs to be a way of preventing too many messages from coming in, when required.

*Akin To:* (specialisation of) Message Filter [77].

**Example:** A CRM printing solution experiences memory failures if batches of more than one hundred letters are sent into it at a time.

**Pattern Batc41: Multi-Channel Batch Receive.** In certain scenarios messages need to be pulled off multiple channels and consumed together. If
a business rule states that message batches must have something in common, the Message Filtering pattern can be combined with this pattern. The problem is that messages that need to be grouped together are from different sources (channels). This pattern can be seamlessly combined with Message-Message Comparison Filter. In this case it is possible to detect that certain a combination/s of messages satisfies a business rule.

Example: In a telecommunications company there are monitoring processes constantly watching for patterns of error messages from various subsystems. Certain combinations of fault messages indicate that an optical-fibre cut has occurred.

5.9 Batch Filtering

The Batch Filtering patterns combine the Filtering patterns with the Batch Receive patterns. The problem is that there may be thousands of messages waiting on a channel, and a select subset of them need to be processed together. This way messages are choosily consumed as one unit by the process.

Pattern BatFil42: Criteria Batch Filtering. The concept is to con-
sume a set of messages that satisfy certain criteria. This solves the problem where only a certain subset of available messages qualify for processing.

_Akin To_: (specialisation of) Message Filter [77], Property-based correlation [25].

_Example:_ A recruitment agency only processes applications where the applicant has tertiary qualifications.

**Pattern BatFil43: Message-Message Comparison Filter.** This pattern uses the data in two or more messages and compares them. If the data in both messages, viewed together satisfy a business rule then the process can receive the messages. The problem is that there may be thousands of messages on a channel (or set of channels). These messages may need to be combined together as a batch to be processed properly, in specific combinations. How can we find these right combinations without consuming the messages and looking at their data?

_Akin To_: SQL Aggregate Functions [82].

_Example:_ Only launch a process for a customer if the sum total of all loan amounts is greater than one million dollars.

**Pattern BatFil44: a Priori Runtime Knowledge Filtering.** Batch filtering allows messages with common property values to be consumed together – when they possess attributes in common with process data. However if the process is not yet started it then has no runtime variable values to compare with messages. There is a strong motivation in the business to group messages together according to common attributes values, however in many situations it is difficult or impossible to know what those common values are. In order to achieve this the workflow system needs to survey the messages on a channel and then look for those with common property values and consume them together – irrespective of what those property values happen to be.

_Akin To_: SQL Group By [82].

_Example:_ Mail sorting centre wishes to start a depot dispatch process each
time a certain number of letters have passed through, all going to the same depot. There is no stipulation about which depot that the batch of letters is addressed to.

Example: 2 All the purchase orders from the same customer need to be invoiced together each month.

**Pattern BatFil45: Better Messages.** The idea is to consume the better/best messages from a set of available messages. The problem is that the benchmark for message selection is not defined, or there is a preference to pick the best for purpose messages, according to some comparable value. This is a specialisation of the Message-Message Comparison Filter where the selected set is based on being better, in comparison to other messages. The way the best is selected should support either/both: (1) selecting the best specified number of messages, (2) selecting the best specified percentage of messages.

Akin To: SQL Aggregate Functions [82], Property-based correlation [25].

Example: A committee overseeing a request for tender process only has time to consider the three quotes with the lowest price.

**Pattern BatFil46: Message Sort.** The idea is to be able to express some ordering constraint over messages being selected on a channel. Ordering messages can assist with message selection or assist with the orderly processing of messages – post consumption.

Akin To: Resequencer [77], SQL Order-By [82].

Example: A recruitment agency wishes to sort job applications by the number of years of experience.

**Pattern BatFil47: LIFO/FIFO Comparison Filter.** The intent is to control which message/s are consumed based on the time that they were put onto their respective channel. A LIFO prefers the most recent message/s whereas the FIFO prefers the least recent ones.
Akin To: SQL Order-By [82], SQL Top Expression\(^7\).

Example: A hotel offers a sixty percent discount to the first thirty five bookings to stay during June.

**Pattern BatFil48: Batch Time-Schedule Comparison Filter.** This pattern typically compares a range of times to a set of message timestamps and consumes any messages that have timestamps occurring within the specified time range. If the timestamps on the messages occur within the specified time range then the messages are included.

Akin To: Time-interval-based correlation [25].

Example: Each day a bank receives millions of transactions to move money from one bank account to another. Many of these accounts are in other banks. A bank automatically processes these inter-bank transactions after hours, starting at the end of each business day. It filters for transactions that were timestamped (inclusively) between 5:00:00.000001 pm yesterday and 5:00:00.000000 pm today.

**Pattern BatFil49: Batch Frequency Spike Filter.** This pattern expresses the ability to only consume messages if a greater amount than some fixed threshold arrive within a certain time-window. In business process integration there are situations where a larger number of messages arriving in a concentrated period of time require special action to be taken.

Example: A telecommunications company have thousands of hardware assets distributed across a wide geographic area. Many of these assets are constantly sending status signals regarding their operational state. The relevant process owners need alerting if there are spikes in the numbers of status messages over a given time frame.

5.10 Transformations

*Pattern Trans50: Process Message < -- > Wire Message.* Messages sent over-the-wire differ, in terms of format, depending on the integration middleware being used. It would indeed be extremely taxing at process design time if we had to worry about these differences. Imagine having to worry about big-endian or little-endian byte ordering for integers, or having to worry about whether the SOAP envelope in our message conforms to the SOAP standard. We should not care about such minutia, at the conceptual level. Supporting over-the-wire transformations is uncommon in process integration systems. Many ESB systems demand over-the-wire detail at the conceptual layer. The minutia of over the wire formats should be taken care of by the process modeling/execution technology, allowing only the relevant content and structure of messages at the modelling level.

*Akin To: Normalizer [77].

*Example:* A courier company establish a new partnership with an interstate courier. This company uses JMS whereas other partner couriers use SOAP.

*Pattern Trans51: Data Transformations.* In the distributed world there are many data formats over the same data. Integrations constantly have to transform the remote data format into a local one. If the local data format contains information that is not in the remote format (e.g. a required country code field in an address) there needs to be a strategy for inserting that missing data during or after transformation.

In order to have consistency of data format in processes it makes a lot of sense to do this transformation before the data is passed into the process [36]. However this requires an interface process.

*Akin To: Normalizer [77].

*Example:* A request for a tender is sent to four trading partners as an email. One trading partner replies with an MS Word attachment sent over email. Another trading partner replies to the quote request with a PDF document attachment. The third partner replies by email as free text. These three formats all need to be presented to the review panel as electronic forms.
5.11 Process Discovery

Process discovery is akin to service discovery. It is the ability of two or more automatic processes to begin coordinating actions where they were not fully known at the time of process design. This translates to a process being able to ‘learn’ of new trading opportunities at runtime. Given that integrating with remote processes - with a great degree of certainty about the state of your partner/s - requires human effort; the field of dynamic process discovery is still in its infancy. We know of two kinds of dynamic discovery.

**Pattern Disc52: Service Registries.** There are well known places where services can be advertised on the Web called service registries. They are essentially a Yellow Pages directory for Web Services. They are an opportunity for a service provider to announce a service to the world and for service users to look up and find the service they require. Standards technologies such as Universal Description Discovery and Integration (UDDI) enable a structured and parsable way for services to be dynamically shared. Approaches that automatically compose services together based on registries have been proposed. For instance Hu et. al. [78] propose a technique for automatically composing services from constituent parts based on subscriptions (of service inputs) and advertising (of service outputs).

Describing a service is non trivial. There are many subtle aspects to any service, such as notions of quality, standards compliance, how long it will be around, voluntary/mandatory warranties, and the legal framework of the country where the service is hosted, are just a few [116]. Indeed the problem of capturing all that information is non-trivial, but perhaps the greater challenge is automatically deciding which is the best fit to our needs. For that reason we would consider this pattern, while interesting, to be a non-core pattern – meaning that it only need be supported once sufficient progress has been made in the fields of artificial intelligence, service discovery and ontologies.

*Example:* A luxury boat builder is looking for an automated financing service provider for its customers.

**Pattern Disc53: Channel Mobility.** Sometimes a business learns about a new collaborator while a process is executing. There needs to be a way to model the sharing and learn about these new collaborators. This pattern expresses that a process forwards contact details it knows about to another process. A process receives the contact details in a message. Having learned
of the new contact it starts communication to it.

Theoretical approaches such as $\pi$-calculus \cite{102} are well known for their ability to conceptualize channel passing. The JMS API supports mobility of JMS channels.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{channel_mobility_step1.png}
\caption{Channel mobility: Step 1.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{channel_mobility_step2.png}
\caption{Channel mobility: Step 2.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{channel_mobility_step3.png}
\caption{Channel mobility: Step 3.}
\end{figure}

Example: A retailer forwards the contact details of a customer to a shipping process. The shipping process then uses this contact to communicate with the customer.

Pattern Disc54: Process Mobility. In this case, a process instance is moved from one process engine to another. The idea is that a process is started in one site, and then during process runtime it gets moved to another site. This could be helpful in patient care scenarios, for example, where a process gets started in one hospital, but due to capacity constraints the patient is moved to another hospital. The process instance for that patient
gets moved to the new hospital, with the patient. Process mobility adds a raft of new technical challenges to core process integration; including migration of process instance, process model, any format translations necessary between sites, peripheral data migration (data that is relevant to a process instance but not strictly part of it – e.g. a patient history). These challenges make process mobility a non-core problem. Consequently it is mentioned here for completeness but is not essential in a process integration language.

Example: A child-care agency worker in the field begins handling three cases using interactions on a laptop. When the worker returns to base those processes and their state are uploaded into the central workflow server.

5.12 Conclusion

In software engineering circles patterns have helped the process of object-oriented analysis and design. For many years, commonly encountered problems can be solved with patterns that express the essence of a design.

The process integration patterns, on the other hand, are not intended to guide the process of process design to such a great degree. Consequently they don’t have long detailed sections in them about how to implement them in YAWL, BPEL. Their first purpose is to shape our thinking about the sorts of problems a process integration needs to solve.

Secondly, selection of a process execution platform is a precarious and difficult task. If there were some expression of what the problems of process integration are then we would at least have some hope of being able to judge alternative process integration platforms based on their technical merits. This set of patterns is essentially inspired by an analysis of different forms of integration middleware; an analysis of middleware aggregation technologies such as Enterprise Service Bus technologies; and an analysis of process integration technologies. No process integration technology is likely to solve all of these requirements. However these patterns could be used to define/select relevant requirements for a process integration project.
Chapter 6

A Language for Modelling Process Integration

This chapter introduces Yet Another Workflow Integration Language (YAWiL). YAWiL is a set of extensions to Yet Another Workflow Language (YAWL) [6]. It supports the process integration patterns: Message Consumption patterns, Request-Response patterns, Correlation patterns, Conversation patterns, Event-Based Process patterns, Filtering patterns, Batch Messaging patterns, Batch Filtering patterns, Transformation patterns, and Process Discovery patterns (see Chapter 5).

YAWiL distinguishes itself from other process integration languages in its emphasis on exposing the interaction architecture (e.g. concerning de/coupling) at the process modelling layer. This is important because YAWiL is intended to be an executable process integration language that is expressive, conceptual, precise, suitable and understandable.

YAWiL supports the (de-) coupling of processes in terms of Threading, Time, and Space. This aspect of integration, having been formally defined in Chapter 3, allows these foundational concepts to be viewed and reasoned about during process design.

Section 6.1 defines the syntax of the extensions to YAWL concerning process integration. Section 6.2 presents some illustrative examples. Section 6.3 shows how YAWiL supports the patterns presented previously in Chapter 5. Section 6.4 presents an in-progress implementation of YAWiL.

6.1 YAWiL Language

This section introduces YAWiL. It begins with Section 6.1.1 which defines simple and complex forms of message sending. Simple and complex forms
of message receiving and filtering are defined in Section 6.1.2. An architecture and implementation for message filtering was described in Chapter 4 and Section 6.1.2 shows how the YAWiL language extensions tie into that architecture. Request Response communication is specified in Section 6.1.3.

Channels represent resources for communication (such as middleware instances) that exist to convey messages across enterprises. Properties represent a resource that can pull atomic data values out of messages flowing along a channel. Channels and properties, as parts of YAWiL, are defined in Section 6.1.4.

Non-blocking communication is essential for many integration patterns, and is defined in Section 6.1.5. Chapter 3 provided full semantics of blocking and non-blocking communication. YAWiL provides a concise syntax for conversation modelling. Conversation modelling enables quickly designing how messages are to be sent to the right process instance. It is presented in Section 6.1.6. Channel mobility is essential for some integrations and is presented in Section 6.1.7. Channels are abstractions of real-world middleware resources and a model showing their relationships is presented in Section 6.1.8.

6.1.1 Send Tasks

The four types of send-task each require a channel parameter (or list of channels) and a message parameter (or list of messages). Both parameters are mandatory.

Send/Reply A Send or Reply is represented by the one type of task (Fig. 6.1(a)). It sends one message onto one channel. The task is finished when the middleware signals that the interaction is complete. If the middleware (and channel) is time-decoupled this would be once the message has been queued. If the middleware is time-coupled the task will complete once the message is received by another endpoint. The Send/Reply task can also send a response or a fault onto a two-way channel (see Fig. 6.2).

Batch Send A Batch Send (Fig. 6.1(b)) sends a batch of messages to one channel, one message at a time. If the channel is time-decoupled the task will complete once the last message has been queued. If the middleware is time-coupled the task will complete immediately after the last message is received by another endpoint.
Multicast  A Multicast (Fig. 6.1(c)) sends one copy of a message to the parameterized list of channels. This list of channels can be dynamically built during process runtime, or it can be coded at design-time by populating the list variable with initial data.

Scatter  A Scatter (Fig. 6.1(d)) sends a list of messages to a list of channels. The first message in the list is sent to the first channel, the second message is sent to the second channel, and so on. If the list of channels is not equal in
length to the list of messages an implementation should generate a runtime error.

**Message Send Model**

The syntax of YAWiL is presented in a series of Object Role Modeling (ORM) diagrams [69]. Figure 6.3 is an ORM diagram for the send communication tasks for YAWiL. However, before going into detail concerning this diagram the following paragraph is a light introduction to ORM.

![ORM Diagram](image)

**Figure 6.3**: Conceptual syntax for message send tasks.

The oval nodes in an ORM diagram (e.g. Figure 6.3) are **Entity Types**. **Entity Types** represent tangible objects or abstract objects in a domain of discourse. The primary reference (identification) scheme for **Entity Types** can be found in parentheses under the Entity Type’s label. The rectangular nodes of the diagram are called **Fact Types**. **Fact Types** represent the relationships between **Entity Types** in the model. The divisions within a **Fact Type** represent various **Roles** played by **Entity Types** in the relationship. The phrases inside or under **Roles** represent **Readings** that help form logical statements about a **Fact Type**. The arrows above a **Role** in a **Fact Type** represent unique constraints over that **Role**. The dot on a connector between an
Entity Type and a Role means that any instance of that entity must play that role (i.e. the role is mandatory). For example in Figure 6.3 the Fact Type between the entity Net and entity YAWL Atomic Task has the following reading: Any instance of a Net must have a YAWL Atomic Task. Furthermore, due to the uniqueness constraint a Net can have many YAWL Atomic Tasks but a YAWL Atomic Task is in only one Net. A bold arrow connecting two Entity Types indicates a subtype relationship. A subtype of an Entity Type inherits all of its Fact Types. In ORM subtypes are defined purely by the relationships they play, thus it is not uncommon to see an ORM sub-type hierarchy that is markedly different to the intuitive type-taxonomy. This can be seen in the subtypes of Figure 6.3. They do not match the send tasks presented graphically in Figure 6.1. A complete description of the ORM language can be found elsewhere [69].

In Figure 6.3 the grey Entities and Fact Types nodes represent syntactical constructs found in YAWL. The YAWL constructs are not the complete syntax for YAWL, they merely present enough detail about YAWL to accurately capture relationships between YAWL and YAWL. A complete definition of the control flow perspective of YAWL is given by van der Aalst and ter Hofstede [6]. Descriptions of the implementation of the data perspective in YAWL is given by van der Aalst et al. [4].

<table>
<thead>
<tr>
<th>Subtype Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Task is a YAWL Atomic Task that is communication task.</td>
</tr>
<tr>
<td>A Single Msg Send is a Communication Task that is of type ( t \in { \text{Send}, \text{Multicast}, \text{Request Response} } ).</td>
</tr>
<tr>
<td>A Single Channel Send is a Communication Task that is of type ( t \in { \text{Send}, \text{Batch Send}, \text{Request Response} } ).</td>
</tr>
<tr>
<td>A Multi Message Send is a Communication Task that is of type ( t \in { \text{Batch Send}, \text{Scatter} } ).</td>
</tr>
<tr>
<td>A Multi Channel Send is a Communication Task that is of type ( t \in { \text{Scatter}, \text{Multicast} } ).</td>
</tr>
<tr>
<td>A Not List Var is a Variable that is not list.</td>
</tr>
<tr>
<td>A Message Var is a Not List Var that has type Message.</td>
</tr>
<tr>
<td>A Channel Var is a Not List Var that has type Channel.</td>
</tr>
<tr>
<td>A List Var is a Variable that is list.</td>
</tr>
<tr>
<td>A Message List Var is a List Var that is list of type Message.</td>
</tr>
<tr>
<td>A Channel List Var is a List Var that is list of type Channel.</td>
</tr>
</tbody>
</table>

Table 6.1: Subtype definitions for send tasks ORM diagram (in Figure 6.3).

Any Communication Task is of type Task Type. A Task Type can take one of nine values ranging from Scatter, through to Property Query Receive. The Task Type corresponds with the name of the communication task (e.g. see Figure 6.1). Note that the subtype names correspond to the
role/s they play and not the task type.

Subtype definitions, over the ORM diagram presented in Figure 6.3, are provided in Table 6.1. The various subtypes of Communication Task need messages and channels to perform their purpose. Instances of Channels and Messages are contained in Net Variables (or lists thereof). A Communication Task can use Channels that are dynamically assigned to Net Variables during process execution. Alternatively, channels can be assigned to variables at process design-time: Channel Var has constant Channel. Channel-List Var has constant Channel in ordering of Number. It is a runtime error for a send/receive task to start when its channel/list variable is empty.

6.1.2 Receive Tasks

Four types of receive tasks are presented in Figure 6.4.

![Receive tasks diagram](image.png)

**Figure 6.4:** Receive tasks.

**Receive** A Receive task (Figure 6.4(a)) fires when one message on one channel satisfies the task’s filter. Input parameters to this task are:

1. Channel represents the JMS-queue/email-inbox/SOAP-operation supplying incoming messages.

2. Filter specifies what message to receive.\(^1\) If unspecified the task will

\(^1\)Section 4.2 provided a conceptualisation of filters.
accept any message on the channel. If the filter matches more than one
message, any one of them is chosen.

3. RemoveFlag is a boolean flag that specifies whether to remove the
message from the channel. This optional flag defaults to true.

4. CopyFlag is a boolean flag. If it is set to true any messages matching
the filter expression will be copied into the receive task’s data space.
This optional flag defaults to true.

The remove flag and copy flag, combined, support the message consump-
tion patterns: Pattern Cons10 Take (RemoveFlag='true', CopyFlag='true'),
Pattern Cons11 Copy (RemoveFlag='false', CopyFlag='true'), and Pattern
Cons12 Clean (RemoveFlag='true', CopyFlag='false'). Note that the lan-
guage also allows the (RemoveFlag='false', CopyFlag='false') combination.

---

Batch Receive  A Batch Receive task (Figure 6.4(b)) fires when a batch
of messages on a channel satisfy, if specified: (i) a filter, (ii) a group property
and (iii) a min constraint. The Batch Receive task is capable of consuming
this message batch and passing it to the process. Batch Receive includes the
parameters of Receive. Unlike Receive, however, it is able to produce more
than one message on its output parameter.

Batch Receive (Figure 6.4(b)) adds input parameters to Receive. These
are described below in their order of evaluation:

1. Min (an optional parameter) prevents the task from completing until
a minimum number of messages can be acted upon. A min number of
messages must satisfy the filter. It can potentially consume more than
the minimum, if the filter allows it to. If unspecified a default value of
min = 1 is applied to the receive. Min is applied immediately after the
filter.

2. Group Property (an optional parameter) enables messages that have
have same property value to be consumed as a group without the need
for the process to know what the common value is beforehand. Group
property supports Pattern 44: a Priori Runtime Knowledge batch fil-
tering. If there are several groups of messages, each sharing a common
property value, one group of messages is randomly selected.

3. Sort Property (an optional parameter) enables sorting the list of mes-
sages being passed into the process. The sorting occurs after applying
the filter. The sort can be specified as forward (ascending) or reversed

163
(descending). Messages are sorted after applying the filter but before applying the max constraint. This enables support for Patterns Batc39 and Batc40 (see Section 5.8).

4. Max (an optional parameter) prevents the task from acting on a set of messages greater than a maximum amount. The max messages are checked after applying the filter. The selection of the subset of messages up to max is non-deterministic. If no max is specified all filter matches are consumed.

Parameters Copy and Remove are evaluated last.

**Multi Receive** A Multi Receive task (Figure 6.4(c)) fires when messages are on many input channels. In order to fire the task, there has be at least one message on each input channel and the filter must evaluate to true. The task, on firing, can consume exactly one message from each channel at the exact same moment. The order of the output message list corresponds to the order of the input channel list.

The input parameters of the Multi Receive are correspond to those of Receive with the exceptions of: (i) the input channel becoming a list of channels and (ii) the output message becomes a message list. The filter expression can refer to message properties belonging to any/all input channels. The task completes once a message is available on each channel has fired it, and it has had the opportunity to consume those messages.

Related messages off different channels can be consumed concurrently (see Pattern BatFil43 Message-Message Comparison Filter) using this task. The direct comparison of two message properties on different channels would be semantically equivalent to an SQL join (see Chapter 4). For instance if channels $c_1, c_2$ had properties $p_1, p_2$ respectively the corresponding SQL join could be:

\[
\begin{align*}
&\text{SELECT } c1.\text{msgid}, c2.\text{msgid} \\
&\text{FROM } c1, c2 \\
&\text{WHERE } c1.p1 = c2.p2
\end{align*}
\]

If more than one message, on a given channel, satisfy the filter then one is chosen to be in the list. Which message gets chosen is not defined.

**Property Query Receive** A Property Query Receive task (Fig. 6.4(d)) fires when its query produces a non-empty set of message-IDs. The task makes use of any query language. It therefore inherits the conceptual power of a full query language. For those scenarios where the constraints of a simplified
filtering syntax is not powerful enough (e.g. see example for Pattern 43 in Section 5.9) this task is useful. The constrained filtering syntax is meant to reduce modelling effort in filter specification, however it does not offer the expressive power of a query language. For instance consider queries that apply aggregate mathematical functions over several messages concurrently. Note that no channels are passed into this task. This is because the channels may be directly referred to in the query, for example:

\[
\text{SELECT msgid FROM "chanLoan" AS cl, (}
\text{SELECT customer}
\text{FROM "chanLoan"
\text{GROUP BY customer
\text{HAVING SUM("amount") >= 1000000
\text{LIMIT 1}
\text{)}AS elegible
\text{WHERE cl.customer = elegible.customer}
\]

Receive Task Model

Figure 6.5 presents the abstract syntax for the receive communication tasks in ORM. Receive tasks are a subtype Communication Task – like send tasks. Also, similar to send tasks, the Task Type of a Communication Task determines the subtype/s a receive task can become. The subtype definitions for these receive tasks are presented in Table 6.2.

In Figure 6.5, every Net has one instance channel var. An instance channel var holds a channel that is created at the same as a net instance. The net can share this channel with remote processes so that they may direct replies back (see Pattern Corr18 Instance Channels).

A process integration model written in YAWiL may refer to many channels. Some of those channels may be used for sending messages and some for receiving. The enterprise environment will contain channels where business rules prevent messages being received by a process (something other than the process has that right). When a channel is added to a YAWiL engine that business rule can be expressed over the channel, defining the send/receive rights it has from the perspective of the process. A Channel must be locally owned (supports inbound) in order to receive messages off it. Attempting to receive over a channel which is not locally owned is a runtime error.

Any Receive Task may have a filter (as shown in Figure 6.6) unless it is an instance of Property Query Receive. A Property Receive Query
Having the right to receive off a channel is a necessary condition to receive a message. For example only if an email address is ours do we have the right to receive mail off it.

Figure 6.5: Conceptual syntax for message receive tasks.

Figure 6.6: Conceptual syntax for message filtering.
Subtype Definitions

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Msg Receive</td>
<td>is a Receive Task that is of type ( t \in {\text{Receive, Request Response}} ).</td>
</tr>
<tr>
<td>Multi Message Receive</td>
<td>is a Receive Task that is of type ( t \in {\text{Batch Receive, Multi Receive, Property Query Receive}} ).</td>
</tr>
<tr>
<td>Single Channel Receive</td>
<td>is a Receive Task that is of type ( t \in {\text{Receive, Request Response}} ).</td>
</tr>
<tr>
<td>Multi Channel Receive</td>
<td>is a Receive Task that is of type ( t \in {\text{Multi Receive, Property Query Receive}} ).</td>
</tr>
<tr>
<td>May Filter</td>
<td>is a Receive Task that is of type ( t \in {\text{Request Response, Receive, Batch Receive, Multi Receive}} ).</td>
</tr>
<tr>
<td>Property Query Receive</td>
<td>is a Multi Message Receive that is of type Property Query Receive.</td>
</tr>
</tbody>
</table>

Table 6.2: Subtype definitions for receive tasks ORM diagram (in Figure 6.5).

must have a Query. The subtype rules in Table 6.2 state that Communication Tasks of type Property Query Receive are not instances of Single Channel Receive or Multi Channel Receive. Thus they do not have any Channel Vars. Instead the query must contain direct references to the channel/s it is receiving from. This is shown in the ORM diagram by the exclusion constraint between has filter and has query.

The filter and the query are not necessarily bound to a particular query language. However the current implementation uses SQL. The ORM diagram shows both a query and a filter expression as a textual type in the model – despite the fact that they are structured. In an SQL-based implementation a query is an SQL query, and a filter is structurally similar to a WHERE-Clause. Chapter 4 established the conceptual link between DB tables and Channels, and between DB Columns and properties. Consequently, for example, a Single Channel Receive receives over a Channel Var. At runtime this Channel Var yields the table name in an SQL FROM-Clause. Furthermore a property reference in a filter corresponds to a database column reference on a table (see Chapter 4). Put succinctly, a filter can only contain a \(<\text{property reference}>\) if the corresponding Property is bound to a Channel that the Receive Task uses. Violation of this rule constitutes an error. The BNF for an SQL Column Reference [82] is refined to include a property reference:

\[
<\text{column reference}> ::= \\
[ <\text{qualifier}> <\text{period}> ] <\text{column name}>
\]

\[2\]The query must select message-id for an implementation to know what messages to provide back to the task (see Chapter 4.3.3).
| <property reference>
<property reference> ::= <identifier>

Process variables must be accessible to a filter (query) in order to support comparison of message contents to process data. This supports for example, Patterns Filt31 Data-Message Comparison Filter, Filt32 Time-Schedule Comparison Filter, Filt33 Time-Timestamp Comparison Filter, BatFil42 Criteria Batch Filtering, BatFil48 Batch Time-Schedule Comparison Filter. Thus the BNF for an SQL Unsigned Literal [82] is extended to contain a Process Variable Reference.

<unsigned literal> ::= <unsigned numeric literal>
  | <general literal> | <process variable reference>
<process variable reference> ::= <dollar><identifier><dollar>
<dollar> ::= $

An example of such an expression is:
$MaxPrice$ > TotalValue

Where $MaxPrice$ refers to a process variable. TotalValue refers to a property (extracted from a message and stored in a database column).

6.1.3 Request-Response

The semantics of request-response interactions are obviously quite distinct from uni-directional messaging. Request-response interactions help raise certainty about the state of a remote process, help with coordinated action, support immediacy, and allow feedback between partners.

Figure 6.7: Request-response task.

Figure 6.7 presents a communication task for supporting request-response style interactions. On the left-hand side of the task is the input message and
on the right is the output response. This task has an exception handling event receive attached to it. Should an exception occur, as a result of the invocation, the process will follow any arcs leading out of the exception handling construct. Such arcs could possibly start up a compensation sequence.

The Filter, Copy Flag, and Remove Flag are optional input parameters to the Request-Response task. They have the same meaning as the similarly-named input parameters of the Receive task.

The use of a request-response task in one process to initiate another process is illustrated in Figure 6.8.

![Figure 6.8: Request-response task in a process.](image)

**Request-Response Model**

According to the subtype defining rules in Sections 6.1.1 and 6.1.2 a Request-Response task is an instance of the following Entity Types: Single Msg Send, Single Channel Send, Single Msg Receive, Single Channel Receive.

A Communication Task of type Request Response must send over and receive over the same Channel Var instance. That Channel Var must be request response. Furthermore any Channel Var that is request response can have constant Channel iff that Channel has interaction type request-response. This means that request-response channel variables can only hold request response channels (and vice-versa). This is a design time constraint and an implementation should throw an error during runtime should the constraint be violated (it could be violated through channel variable assignment).
A Request-Response task is not the only task that can connect with a request-response channel. A receive task can listen on a request-response channel, provided that it is ultimately followed by a Send/Reply task in the process. This is necessary to model a request-response service. An example showing this can be seen in Figure 6.8. This leads to the following constraint over the abstract syntax: Any Communication Task of type receive that receives over a Channel Variable that is request response must be followed (in the process flow) by a Communication Task of type send.

6.1.4 Channels and Properties

The extensions to YAWL for process integration are layered over three foundational concepts. These are Channels, Properties and Filters. Filters are explained in Section 6.1.2.

Channels represent middleware resources for transporting messages.

Properties represent selected parts of a message that a process is interested in.

Previously, Chapter 3 provided an in-depth formal semantics for channel coupling. Chapter 4 provided an in-depth description of properties, channels and filters. Sections 6.1.1 and 6.1.2 defined how communication tasks use channels, properties and filters. This section shows how channel and property instances, are modelled.

Channels

A channel can be uni-directional (e.g. email) or bi-directional (e.g. SOAP-operation). A channel can be time-coupled (e.g. RMI) or time-decoupled (e.g. JMS). These qualities of a channel are usually determined by the type of middleware managing them. There are alternative coupling semantics that a uni-directional channel can be modelled with. These are shown in Figure 6.9.

Figure 6.9 presents fundamental properties a one-way channel can have. Time-decoupled channels are adorned with a surrounding dotted circle (Fig. 6.9(d)). A circle was chosen because time-decoupling is semantically related to a Petri-net place (see Section 3.6(b)). Space-decoupled channels are represented with a wavy structure dividing the channel through the middle (Fig. 6.9(b)). Space-decoupled-topics are represented with a wavy structure dividing one pipe in and several pipes out (Fig. 6.9(c)). Space-coupled channels are represented without the wavy dividing structure (Fig. 6.9(a)).
Time-coupled channels are represented without surrounding dotted circles (Figs. 6.9(a), 6.9(b), 6.9(c)).

There are several ways that request-response channels can be modelled in terms of their coupling. These are shown in Figure 6.10. Fig. 6.10(a) models a time-coupled, space-coupled, request-response channel. Figure 6.10(b) models a time-decoupled, space-coupled, request-response channel. These can model any imaginable coupling combination. For instance consider a space-decoupled/publish-subscribe request-response channel (e.g. a JMS Topic Requestor\(^3\)). Such a coupling configuration can easily be expressed using the adornments presented in Figure 6.9.

Variables are used for referring to a channel in any YAWiL model. Most

channels are incorporated into a process at design time. These are channel constants. In some scenarios channels can be discovered during process execution. For those cases the channel variables, and channel variable lists (see Figure 6.11) are modelled in dotted notation.

Channels are consistently used by more than one process. For this reason they are declared outside of the YAWiL process. When they are declared there are some additional attributes needed beyond the graphical adornments already presented (see Figure 6.12(a)). Attribute Name uniquely identifies the channel (e.g. purchase-order@acme.com). The CommunicationAdapter attribute identifies an object or resource capable of providing connectivity (binding) to a middleware technology. For example, an email communication adapter would be an object capable of connecting with a specific email server in an organisation. The DataType attribute allows us as process modellers to define a data type of messages we expect to find passing through the channel. It is an error to send a message on a channel if that message violates the channel’s datatype. Such a bad message should be placed on a dead letter queue. If the channel is already declared in WSDL an alternative set of attributes is permissible – allowing the channel to be expressed with three literals: (i) a WSDL-URL, (ii) a PortType and (iii) an OperationName (see Fig. 6.12(b)).
Properties

Properties extract values from messages once they arrive into the process execution context (e.g. onto the JCoupling engine). The values they extract from messages are recorded to a persistent store (see Chapter 4). Properties enable communication tasks to filter any inbound message while abstracting the process layer from differences in the structure, the format, and the transportation of the message. Filters refer to properties, not messages. The property deals with format differences of the message encapsulating this concern from higher modelling layers.

Properties are modelled graphically as a magnifying glass looking at a message (see Fig. 6.13). Five property templates are presented, however more can be created. The XPath property template (Fig. 6.13(a)) can extract values from XML messages and is modelled with a target Channel, Label, ResultType and an XPathExpression. All properties have a label used in the filtering expressions. The Timestamp property (Fig. 6.13(b)) can record the time of arrival of a message. The SOAP-Header property (Fig. 6.13(c)) can

Figure 6.13: Properties can be extended to extract values from any conceivable message type/format. These are some examples.
extract named header values out of an inbound message and is modelled with
a target Channel, Label and HeaderName. HeaderName matches the name
of the message header to extract. The Channel Extract Property template
(Fig. 6.13(d)) is able to extract references to new channels out of incoming
messages and is modelled with a target Channel and an XPathExpression.
The Compound Property template (Fig. 6.13(e)) allows multiple properties
to be treated as one property. This is useful for Compound Property-Based
Conversations (see Pattern Conv22). Properties, have a label, an output
result type, and a source channel. Certain property types, due to their nature
(i.e. the Timestamp Property) have a fixed result type. Property constructs
may also add any number of additional attributes to help define exactly how
to retrieve values from incoming messages. For example the XPath property
requires the XPath Expression attribute (XPathExpr). As shown properties
are linked to a channel.

Properties and channels abstract the IT communication resources in an
enterprise network. They provide a neat way of viewing and comparing the
contents of many messages within the same channel, or in many channels,
regardless of transport, message structure and storage format. Channels,
typically, do not belong to any particular process instance; they are shared
between many process instances, much like the underlying communication
resources of the enterprise network.

Channels and Properties Model

Figure 6.14 is an ORM diagram defining the syntax for Properties and their
relationships with Channels. Again, (like in Figure 6.3) the grey Entities and
Fact Types nodes represent syntactical elements found in YAWL.

The Process Container could be a a YAWL engine. It can hold many
YAWL Process models, and it can also have many Channels. As men-
tioned previously the YAWL process refers to Channels through variables,
but channels are declared as part of the process container. The Process Con-
tainer could also be useful in migrating a process integration model from one
server to another. This is because it can contain many process models and
all of the Channels and Properties used by them.

Any Channel can have a msg schema. This is the way to declare a
typed channel. Messages on a typed channel must conform to such a type
constraint. Request-Response Channels are a sub-type of Channel\(^4\) and can
have a type constraint declared against the response if desired.

A Channel can have many Properties. This is expressed through the

\(^4\)The sub-type constraint is declared in Figure 6.23.
nested Entity Type Binding. For any message arriving on a Channel all Properties that are bound to the Channel will be executed. It is possible to declare that the property is to be executed on the response leg of a request-response interaction using the role for response. A Binding may only be for response if the Channel is a Request Response Channel.

A Property is an Entity that can only be bound to one Channel – as shown by the uniqueness constraint over role bound to. Any Property must have a label – as shown by the mandatory constraint (dot) on this role. A property can have any label, but no two properties can have the same label for the same channel – as shown by the external uniqueness constraint (denoted as a 'U') on the respective roles. Note that when Properties are referred to in the previously mentioned Filters – they are always referred through their Channel and Label.

As discussed earlier (and in Section 4.3) a property can output typed values. For instance a Property of type TimeStamp outputs a DateTime and a Property of type Channel outputs a Channel. A property can also be designed to output types according to its DataType configuration. An XPath Property, can extract values of a nominated DataType.

Any Property must be either bound to a Channel or contained by a

Figure 6.14: Conceptual syntax for Channels and Properties.
**Property.** This concept is shown graphically in Figure 6.13(e) and is depicted in ORM (Figure 6.14) as a disjunctive mandatory constraint between the roles bound to and contained by.5

Presented in Table 6.3 are the subtype definitions for the ORM Diagram in Figure 6.14.

<table>
<thead>
<tr>
<th>Subtype Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>An Atomic DataType is a DataType that is atomic.</td>
</tr>
<tr>
<td>A Dynamic Type Property is a Property that is of type XPath.</td>
</tr>
<tr>
<td>A Has XPath Property is a Property that is of type ( t \in {XPath, Channel} ).</td>
</tr>
<tr>
<td>A Header Property is a Property that is of type SOAP Header.</td>
</tr>
<tr>
<td>A Request Response Channel is a Channel that has interaction type Request-Response.</td>
</tr>
</tbody>
</table>

Table 6.3: Subtype definitions for the entities presented in Figure 6.14.

### 6.1.5 Non-Blocking Communication

The communication constructs presented thus far cover blocking communication. Non-blocking communication enables a raft of additional capabilities. Capabilities that are essential for performing patterns: Non-blocking Send (Coup2), Non-blocking Receive (Coup4), Receive response: non-blocking (Resp17), Event-based Process Instance Creation (Even24), Event-Driven Task (Even25), And/Or/XOR Event-Driven Task (Even26), Unsolicted Events (Even28), Event-based Task Interruption (Even29), Timeout (Even30).

Non-Blocking communication can be represented in YAWiL by attaching any communication task to a YAWL condition (see Figure 6.15). The first two examples show receive tasks attached to conditions that have no preceding arcs (Figs. 6.15(a) and 6.15(b)).7 Conditions like these (with no preceding arcs) produce a token when an interaction has occurred. The last example in Figure 6.15(c) has preceding arcs. The communication task attached to such a condition gets fired when a token arrives on one of the incoming arcs. The token is then free to continue moving through the process before the communication task is complete (hence non-blocking). Once the attached communication task completes it adds any data to the data space of the YAWL net without affecting the flow of any tokens.

---

5A disjunctive mandatory constraint is captured by a complex mandatory constraint (a dot over two roles) and an exclusion constraint (an ‘X’) over the same roles.

7YAWiL extends YAWL to allow conditions other than input conditions to have no preceding arcs. However such conditions must be attached to a communication task.
Binding a receive task to an input condition means that the receive task eagerly listens for incoming messages. When a message arrives it adds a token to the input condition, meaning a process instance has been created. Binding a receive task to a condition without any inflowing arcs (e.g. Figure 6.15(b)), means that, again, the task eagerly listens for incoming messages. When a message arrives a token is placed into the condition. Such a task is only eagerly listening for messages whilst there exists an active instance of such a process. This is, in essence, a non-blocking receive. Such a design enables event-driven processes that can respond and adapt to outside stimuli.

An event-driven input condition is shown as part of a process in Figure 6.16. Figure 6.17 shows a semantically equivalent process that doesn’t use any non-blocking constructs. This conversion puts all graphical nodes on a directed path from the input condition to the output condition. Such a conversion allows workflow analysis techniques to be conducted on the model. This is due to the fact that the workflow analysis techniques employed in the YAWL toolset exploit the fact that every graphical node in the model must be on a directed path from the start condition to the finish condition.
Figure 6.17: A semantically equivalent process (to that in Figure 6.16) without using an event-driven input condition.

The net in Figure 6.17 (showing semantics of Fig. 6.16) indicates that when a process is started the non-blocking receive task (“Rcv Event”) becomes enabled. Should a message be received one token is flowed into the workflow model and another token is flowed back, re-enabling the receive-task (“Rcv Event”). When a token reaches the end of the workflow any tokens are removed from the receive task. With the non-blocking receive there is no way to control the number of incoming events (tokens) being generated by “Rcv Event”. Therefore we can conclude that this type of construct would lead to workflows where soundness, specifically proper completion, is impractical to guarantee. A solution that is able to deal with this risk remains work to be done.

Non-Blocking Model

The ORM diagram on Figure 6.18 presents the conceptual syntax for non-blocking communication.

Any YAWL Atomic Task either has pre-/postset or is condition bound – it cannot be both. This is shown by the disjunctive mandatory constraints (i.e. exclusion constraints and mandatory constraints) between has pre-/postset and is condition bound. This rule prevents condition bound tasks from participating in the regular flow of the process. Rephrased, the regular flow of the process is performed by the conditions of condition bounds tasks.

A net is considered finished once the output condition is reached [6]. Binding a receive task onto an output condition would violate this rule. Therefore a Condition Bound Receive Task can only be bound to a Has Postset condition. The only condition without a postset is the OutputCondition.

As discussed previously, binding a communication task onto an input condition means that the task starts attempting communication without the
Condition Bindings

Any Condition (id)

No Preset

Has Postset

receives using

YAWL Atomic Task (id)

has preset / has postset

has postset / has preset

YAWL Condition

Has Postset

receives using

Has Preset

sends using

YAWL Communication Task

Condition Bound Send Task

Condition Bound Receive Task

Condition Bound Request Response Task

Receive Task

Net (name)

has input condition

not input condition

X

Condition Bound Request Response Task

request responds using

X

is condition bound

X

is condition bound

X

is condition bound

Figure 6.18: Conceptual syntax for condition bindings.

trigger of a process token. This is apt for receive tasks however not for send tasks. A net ought to be started by an external event. Allowing a non-blocking send task on an input condition violates this principle, as it would eagerly start sending a message without external triggers. If it were allowable to bind a send task onto an input condition the semantics would dictate that such a task would send messages an infinite number of times, thereby creating an infinite number of new process instances. Consequently, a Condition Bound Send Task can only be bound to a Has Preset condition. This prevents a send task binding onto an input condition because it has no preset.

An input condition in YAWL has no incoming arcs. The condition in Figure 6.15(b) also contains no incoming arcs. This is an extension to YAWL. Such a condition must have a receive task bound to it. Thus, any No Preset condition either is input condition or is not input condition. Those No Preset conditions that are not input conditions must receive using a Condition Bound Receive Task. This is shown by the compound mandatory constraint, the exclusion constraint, and the subset constraint over not input condition. This extension of YAWL allows conditions other than the YAWL Input Condition to have no preset. It also means that the input condition may have a receive task bound to it, but any other no preset condition must have one.

Subtype constraints for entities that extend YAWL in Figure 6.18 are shown in Table 6.4.
### Subtype Definitions

| Condition Bound Receive Task is a Receive Task that is condition bound and is of type $t \in \{ \text{Receive, Batch Receive, Multi Receive, Property Query Receive} \}. |
| Condition Bound Request Response Task is a Communication Task that is condition bound and is of type Request Response. |
| Condition Bound Send Task is a Communication Task that is condition bound and is of type $t \in \{ \text{Scatter, Multicast, Batch Send, Send} \}. |

Table 6.4: Subtype definitions for the entities presented in Figure 6.18).

### 6.1.6 Modelling Conversations

While long-standing integrations (involving many interactions) can be modelled using tasks and filters, ensuring model correctness requires significant effort. Consequently a conversation construct is introduced into YAWiL. A conversation construct significantly reduces the modelling effort and technical risk required. It is graphically presented as a label in a box that binds onto a set of related properties. Figure 6.19 shows a process integration using two conversations L.I. and P.O. All properties a conversation binds to are checked for values during the long-standing interactions. The values are then used to direct messages to the right process instances. The point in the process where the values are initialised for the conversation are indicated with a star *. From that point onwards, throughout the conversation, property values for messages are equal. Consequently, as messages flow past any of the four RequestID properties in Figure 6.19, those that yield the same property value are deemed as partaking in the same conversation.

In Figure 6.19 the conversation P.O. is created when the customer sends a Purchase Order to the Retailer. The Retailer iterates over each line item in the purchase order, creating many instances of conversation L.I.. In more detail: different line items are stocked by different suppliers (Supplier A or B). Supplier A or B’s channel is assigned to the channel variable by task Organise Line Items. The choice of supplier is determined by a business rule. The assigned channel is used to send the message that launches another line item conversation L.I. with the supplier. This inner conversation is scoped within the P.O. conversation and thus refers to both the line item and the purchase order properties.

Conversations can be nested (i.e. they can overlap). There is a Conversation Nesting/Overlap Pattern Conv23 mentioned in Chapter 5. The notations supporting nested conversations in YAWiL ensure significant reduction in model complexity over channels and filters. The succinct syntax also reduces the risk or modelling errors. In Figure 6.19 conversation L.I. is...
nested within conversation P.O. because two RequestID Properties are connected into both conversations. Note that the conversation for each line item (L.I.) is in a loop. For each inner conversation instance the value for the RequestID Property is the same, while the value for the LineItem Property changes.

**Conversation Model**

Modelling a conversation is meant to be simple, hence the syntax in Figure 6.20 is concise. A Conversation uses many Properties. Any Conversation must use at least two Properties – as shown by the frequency constraint on role uses.

When two or more properties that are bound to the same channel are used in one conversation the conversation value is the combined value of both properties. An example of such a case is Properties RequestID and LineItem on conversation L.I. in Figure 6.19. Such syntax is useful for the
Compound Property-Based Conversations Pattern Conv22 and the Conversation Nesting/Overlap pattern Conv23. Thus there is way for a conversation to be modelled over more than one property of the one message. When two or more properties (dual properties) are used on each message in a conversation (e.g. in conversation nesting scenarios) the dual property values need to match other dual property values. Defining a sequence over dual property values help remove ambiguity of how the match is executed. Not shown in Figure 6.20 is that a ConversationProperty must have a sequence iff the same Conversation uses two or more Properties that are in turn bound to the same Channel.

6.1.7 Channel Mobility

Having introduced the YAWL language extensions this section introduces an additional type of task (i.e. a custom YAWL service [4]) that enables the Channel Mobility (see pattern Disc53). Thus, the channel mobility of the π-calculus can be supported.

Figure 6.21: Channel mobility.

Figure 6.21(a) presents a task able to extract a channel from a message. The output of the task is a channel, which can be assigned to a channel variable (see Figure 6.11) and used. Figure 6.21(b) presents a task that inserts a channel into a newly created blank message. This is useful in channel mobility scenarios where the sender is trying to disseminate channel information,
thus complementing Figure 6.21(a). Figure 6.22 presents two processes using these tasks in a channel mobility scenario.

The ability of the property to extract the channel from the message is the responsibility of the property. A property could be designed to extract a channel from any message format without affecting the YAWiL process model. The Channel Extraction task, extracts out the channel using its input Property and then adds the Channel to the Process Container (see Figure 6.14).

**Instance Channels**

Implementations of YAWiL should support a channel dedicated to each Net instance (see Instance Channels pattern Corr18). This would allow each Net instance to listen for messages on its own unique communication resource.

Figure 6.22: Processes exploiting instance channels for correlation.

Instance channels remove the need to correlate many interactions between many process instances that all use shared channels. Like channel mobility, the first interaction contains a message which in turn contains the channel. This channel is the instance channel of the process that started the interaction. All replies back to that process instance use the instance channel. This

183
is conceptually aligned with the architecture of the Web, where each resource (process instance) has its own URI (channel).

The model for instance channels was previously captured in ORM. In Figure 6.5 every Net has instance channel Channel Var.

Figure 6.22 shows how process instance $A'1$ adds its instance channel into the message variable $aMessage$. Process $B$, in turn will respond to the quote request. The task “Discover Channel” extracts the instance channel from the message and assigns it to a channel variable. The task “Quote” uses this channel to return the quote.

### 6.1.8 Grounding

Thus far channels have been introduced as abstract message pathways. These channels abstract away from technology and location details, but preserve the interaction architecture by revealing coupling aspects of the middleware at the modelling layer. Time-coupling was defined in Section 3.3.2, Space-coupling was defined in Section 3.3.3, and whether the channels are unidirectional or request-response was defined in Section 3.5.\(^8\)

A process integration model may require using more than one type of middleware. For instance it may need to integrate processes using a combination of SOAP, Websphere MQ, and FTP.

For each instance of a middleware that any process integrates with, there will need to be a middleware adapter. A middleware adapter is essentially a software resource deployed into a process engine that is able to connect with, and send/receive messages over a middleware. If the middleware requires usernames and passwords, server location data etc. the middleware adapter knows this. For instance if a process uses a channel called $process@acme.org$, an email Middleware Adapter will be invoked each time a process sends or receives an email over it.

The ORM diagram in Figure 6.23 presents how to declare the coupling metadata for each channel. As shown every Channel must have a time coupling $t$ in \{time-coupled, time-decoupled\}. Likewise, every Channel must have a space coupling $s$ in \{direct-addressed, space-decoupled, space-decoupled (Pub-Sub)\}. These capture the time and space coupling semantics.

Every Channel may have one adapter Middleware Adapter. Every Middleware Adapter must use one Middleware Technology. For in-

---

\(^8\)Channels may be thread-coupled or decoupled, depending on the middleware they use (see Table 3.6.1) however the support for thread coupling-decoupling within YAW\textsubscript{L} process models (See Section 6.1.5) negates the need to capture that capability in the middleware metamodel for YAW\textsubscript{L}. 184
stance a channel process@acme.org has adapter AcmeEmailAdaptor which uses middleware technology Email. Note that a channel is not required to use a middleware adapter. Any such model with one or more channels not bound to a middleware adapter are abstract (not executable) models. This is because they don’t have a middleware to interact over.

The capabilities of a middleware technology are stable and define what a channel may or may not do. A Middleware Technology supports at least one of \{ time-coupled, time-decoupled \} interactions and it supports at least one of \{ direct-addressed, space-decoupled, space-decoupled (Pub-Sub) \} interactions.

There are three equality constraints over a set of joins in Figure 6.23. The joins are over Middleware Adapter, Time Coupling, Space Coupling and Interaction Type; and these joins are onto Channel and Middleware Technology. The subset constraints mean that a Channel cannot perform any type of space-coupling, time-coupling, or interaction-type that is not
supported by the Channel’s corresponding Middleware Technology.

6.2 Introductory Examples

In this section several introductory examples for illustration purposes. The first example is a simple flight check-in and boarding process, and the second is a sequence of variations over a simple procurement process. Section 6.3 will provide more process integration models, where it is shown how the language supports the process integration patterns.

6.2.1 Flight Check in boarding process

![Diagram of Flight Check-in Process]

Figure 6.24: Flight check in process (repeated from Figure 1.3).

The simple flight check-in and boarding process in illustrated in Figure 6.24. When a flight is ready for check-in a scheduling system sends an event to a non-blocking task “Checkin ready” which instantiates a check-in process. The process combines human-driven action with tasks that integrate other airline information systems.

Passengers can only be checked-in until 30 minutes before departure time. However, because the airline is eager to fill up empty seats, if the flight is delayed and not departed they want to re-open check in. This business rule
is modelled by the non-blocking task “Flight Delay” which triggers a flow of events that can reset the state of any process that is not yet flight ready. Before resetting the flight boarding time a planner determines changes to the flight plan and a gate crew member alerts the passengers. Then a signal is sent to the “Scheduling Update” channel to change the departure time being displayed for that flight by task “Extend book/check window”. The task “Extend book/check window” resets the process back to the condition “Book/Check OK”. Note that the cancelation set of task “Extend book/check window” removes tokens from any part of the workflow and then it puts a token into the desired condition.

The process proceeds again, along the same lines. Provided that no more delays occur the passengers will be boarded by the ground crew, and the process token will reach the output condition. Once the flight is boarded the “Scheduling Update” channel is notified that the flight has departed.

6.2.2 Procurement Process

Procurement processes involving larger amounts of money adopt a tender-quote process, where the best quote is selected. An internal requisition order arrives in the purchasing department, and this triggers off a process instance. Three partners are asked to provide a quote and the best quote is selected. A purchase-order gets sent to the selected partner. This example is presented in several variations illustrating filtering, conversations, and instance channels.

![Diagram of quote selection process](image)

Figure 6.25: Purchase order selection assuming asynchronous quoting services.

Figure 6.25 presents a quote selection process. The request comes in to the procurement department through task “Incoming Order”. This launches
a process, and three remote suppliers are invoked to provide a quote. The requests for quote are authored by the “Build Documents” task. Task “Request Quotes” sends an email to “foo.com”, performs an FTP put to “bar.com”, and sends an XML Document over SOAP to “abc.com”. Three different channels are shown for each supplier. As shown each supplier uses a different middleware technology.

Because the responses can take days, or perhaps weeks, we need to correlate the quotations back to the right local process instance. In the case of Fig. 6.25 all incoming quotes get sent to one “quote” channel, regardless of the process instance. The filter attached to the task “Receive Quotes” defines that incoming quote messages must produce a property value that matches the process variable $\text{Request}$. There is a two day wait until the receive task checks for incoming quotes and receives all that match the filter.

![Diagram](image)

Figure 6.26: Asynchronous purchase order selection using conversation-based filtering and timeout pre-event.

The model presented in Figure 6.26 shows how to achieve the same purpose with less modelling effort (as opposed to Fig. 6.25). The task “Request Quote” sends a message to a list of channels. This list of channels may have been passed in dynamically through the process inputs. The conversation “C” is used to connect the two properties, both called “RequestID”. The conversation ensures that messages leaving and entering the process are correlated, without needing to record correlation IDs in a process variable and then use them in filtering expressions. The “Min : 2” constraint on the receive task ensures that the receive task cannot continue until the first two quotes are received. So the process no longer **Waits 2 Days** for all quotes (as opposed to the process in Fig. 6.25). This modification guarantees that the
process will wait until at least two quotes are received – irrespective of the
time required for them to arrive.

Figure 6.27: Asynchronous purchase order selection using instance channel mes-
saging and timeout pre-event.

Figure 6.27 captures a similar scenario again, but this time using the in-
stance channel as opposed to filters or conversations (see Figs. 6.25 and 6.26).
The instance channel being used removes the need to correlate responses.
Every outgoing request gets stamped with the channel on which to send back
replies by the task “Insert Instance Chan”. In process models instance chan-
nels are depicted with an “i”. This time only the first message to arrive gets
used.
6.3 Patterns in YAWiL

In order to demonstrate the effectiveness of YAWiL at modelling process integration a series of samples are presented in this section. The samples are essentially vignettes expressing one or more patterns in YAWiL.

6.3.1 Coupling Patterns

Each of the coupling patterns are listed in the following table, along with a reference to an example showing how that pattern is expressed in YAWiL.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Send</td>
<td>Coup1</td>
<td>Figure 6.28(a)</td>
</tr>
<tr>
<td>Non-blocking Send</td>
<td>Coup2</td>
<td>Figure 6.28(b)</td>
</tr>
<tr>
<td>Blocking Receive</td>
<td>Coup3</td>
<td>Figure 6.28(a)</td>
</tr>
<tr>
<td>Non-blocking Receive</td>
<td>Coup4</td>
<td>Figures 6.28(b), 6.37 and 6.29(b)</td>
</tr>
<tr>
<td>Time-Coupled</td>
<td>Coup5</td>
<td>Figure 6.28(b)</td>
</tr>
<tr>
<td>Time-Decoupled</td>
<td>Coup6</td>
<td>Figure 6.29(a)</td>
</tr>
<tr>
<td>Space Coupled</td>
<td>Coup7</td>
<td>Figure 6.29(a)</td>
</tr>
<tr>
<td>Space-Decoupled</td>
<td>Coup8</td>
<td>Figure 6.28(b)</td>
</tr>
<tr>
<td>Space-Decoupled Topic</td>
<td>Coup9</td>
<td>Figure 6.28(a)</td>
</tr>
</tbody>
</table>

Table 6.5: Coupling patterns.

(a) Blocking Send, Blocking Receive, Space-Decoupled Topic (i.e. Publish-Subscribe).

(b) Non-blocking Send, Non-blocking Receive, Time-coupled, Space-Decoupled.

Figure 6.28: Coupling patterns.
6.3.2 Message Consumption Patterns

The message consumption patterns are listed, along with a reference to an example in YAWL.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Take</td>
<td>Cons10</td>
<td>Figure 6.30(a)</td>
</tr>
<tr>
<td>Copy</td>
<td>Cons11</td>
<td>Figure 6.30(b)</td>
</tr>
<tr>
<td>Clean</td>
<td>Cons12</td>
<td>Figure 6.30(c)</td>
</tr>
</tbody>
</table>

Table 6.6: Message consumption patterns.

(a) Take all messages.  (b) Copy all messages.  (c) Clean all messages.

Figure 6.30: Controlling message consumption in YAWL.
6.3.3 Request Response Patterns

The request response patterns are listed with a reference to an example in YAWiL.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgement</td>
<td>Resp13</td>
<td>Figure 6.31(a)</td>
</tr>
<tr>
<td>Response</td>
<td>Resp14</td>
<td>Figure 6.31(b)</td>
</tr>
<tr>
<td>Throw Fault</td>
<td>Resp15</td>
<td>Figure 6.31(c)</td>
</tr>
<tr>
<td>Receive response: blocking</td>
<td>Resp16</td>
<td>Figure 6.31(a)</td>
</tr>
<tr>
<td>Receive response: non-blocking</td>
<td>Resp17</td>
<td>Figure 6.31(d)</td>
</tr>
</tbody>
</table>

Table 6.7: Response patterns.

(a) Process A is performing a blocking request-response. Process B is acknowledging the request before processing it.

(b) The request has been processed before sending the response.

(c) The fault is sent from the responder, which triggers a compensation flow in the requestor.

(d) The requestor does not wait for the response.

Figure 6.31: Request response patterns.
6.3.4 Composed Interaction Patterns

The composed interaction patterns (patterns for correlation and conversations) are listed with a reference to a YAWiL example.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance Channels</td>
<td>Corr18</td>
<td>Figure 6.32</td>
</tr>
<tr>
<td>Request-response Correlation</td>
<td>Corr19</td>
<td>Figure 6.31(b)</td>
</tr>
<tr>
<td>Token-based Correlation</td>
<td>Corr20</td>
<td>Figure 6.33</td>
</tr>
<tr>
<td>Property-based Conversations</td>
<td>Corr21</td>
<td>Figure 6.34</td>
</tr>
<tr>
<td>Compound Property-Based Conversations</td>
<td>Corr22</td>
<td>Figure 6.35</td>
</tr>
<tr>
<td>Conversation Nesting/Overlap</td>
<td>Corr23</td>
<td>Figure 6.36</td>
</tr>
</tbody>
</table>

Table 6.8: Composed interaction patterns.

Figure 6.32: In process instance $A'1$ task Request Quote sends its instance channel (denoted with an ‘i’) to process $B$ task Receive Request. Process $B$ discovers how to contact process $A$ in task Discover channel and assigns it to variable. The quote message is sent back over process instance channel belonging to $A'1$. 
Figure 6.33: Process **Buyer** and **Seller** both use a variable **CorrToken** which is inserted in incoming/outgoing messages in order to correlate messages to correct instances. To assist illustration the usually invisible data variable **CorrToken** is depicted graphically. The data perspective of YAWL has no graphical presence. The variable can be inserted into a message using XPath and/or XSLT. Task **Receive Quote** uses a filter to compare message data with process data.

Figure 6.34: Process **Buyer** and **Seller** correlate incoming and outgoing messages. The nodes on the diagram located between the two processes are scoped to the **Process Context**. Conversations, Channels and Properties are declared within the Process Container (see the ORM definitions in Figs. 6.14 and 6.20).
Figure 6.35: In this iteration of the quote scenario the customer’s quote reference is not enough to uniquely define the conversation – in particular for the seller. The customer ID needs to be combined with the customer’s quote reference in order to define the conversation. This is depicted using a compound property at each stage of the conversation.

Figure 6.36: Nested within the conversation between the customer and retailer is the conversation between the retailer and each supplier. The inner conversation P.O. uses property values learned from both the outer and inner conversations by using a compound property. A more complete explanation of this example is provided in the section on modelling conversations (Section 6.1.6).
6.3.5 Event-Based Process Patterns

The event-based patterns are listed with a reference to a YAWiL example.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-based Process Instance Creation</td>
<td>Even24</td>
<td>Figures 6.31(a) and 6.31(b)</td>
</tr>
<tr>
<td>Event-Driven Task</td>
<td>Even25</td>
<td>Figure 6.37</td>
</tr>
<tr>
<td>And/Or/XOR Event-Driven Task</td>
<td>Even26</td>
<td>Figures 6.38(a) and 6.38(b)</td>
</tr>
<tr>
<td>Event-Driven Choice</td>
<td>Even27</td>
<td>Figure 6.39</td>
</tr>
<tr>
<td>Unsolicited Events</td>
<td>Even28</td>
<td>Figure 6.37</td>
</tr>
<tr>
<td>Event-based Task Interruption</td>
<td>Even29</td>
<td>Figures 6.31(c) and 6.37</td>
</tr>
<tr>
<td>Timeout</td>
<td>Even30</td>
<td>Figure 6.39</td>
</tr>
</tbody>
</table>

Table 6.9: Event-driven patterns.

Figure 6.37: An external credit agency may send a message warning that the applicant has a risk of defaulting. This is not solicited (waited for). Receiving a warning causes the event-driven task **Cancel Loan Appl** to cancel the loan.

(a) A courier company changes delivery priority if a user submits a form or the system sends a change event. (b) A travel booking can only be canceled before it is committed. A cancel request being received will put a token into the **Cancel Request** condition. If a token reaches the other input condition **Cancel Booking** will fire.

Figure 6.38: And/Or/XOR event-driven tasks.
6.3.6 Filtering Patterns

The filtering patterns are listed with a reference to a YAWiL example.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data-Message Comparison Filter</td>
<td>Filt31</td>
<td>Figure 6.33</td>
</tr>
<tr>
<td>Time-Schedule Comparison Filter</td>
<td>Filt32</td>
<td>Figure 6.40</td>
</tr>
<tr>
<td>Time-Timestamp Comparison Filter</td>
<td>Filt33</td>
<td>Figure 6.44(b)</td>
</tr>
</tbody>
</table>

Table 6.10: Filtering patterns.

Figure 6.40: A courier company offers a service to any of its customers that allows them to change delivery priority or final delivery address. Change requests made after 10:00 AM are applied at the beginning of the following business day. Each day the next deadline and next morning dates are calculated. Requests to change the delivery address arrive on the channel. The receive task consumes messages as long as they conform to the business time constraints and routes received messages to the Update Delivery subprocess in order to change the delivery address.
6.3.7 Batch Messaging Patterns

The batch messaging patterns are listed with a reference to a YAWiL example.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicast</td>
<td>Bate34</td>
<td>Figure 6.41(a)</td>
</tr>
<tr>
<td>Scatter-Cast</td>
<td>Bate35</td>
<td>Figure 6.41(b)</td>
</tr>
<tr>
<td>Message Batch</td>
<td>Bate36</td>
<td>Figure 6.42(a)</td>
</tr>
<tr>
<td>Batch Send</td>
<td>Bate37</td>
<td>Figure 6.42(b)</td>
</tr>
<tr>
<td>Batch Receive</td>
<td>Bate38</td>
<td>Figure 6.42(c)</td>
</tr>
<tr>
<td>Minimum Batch Receive</td>
<td>Bate39</td>
<td>Figure 6.42(d)</td>
</tr>
<tr>
<td>Maximum Batch Receive</td>
<td>Bate40</td>
<td>Figure 6.42(e)</td>
</tr>
<tr>
<td>Multi-Channel Batch Receive</td>
<td>Bate41</td>
<td>Figure 6.42(f)</td>
</tr>
</tbody>
</table>

Table 6.11: Batch patterns.

(a) Send the request for quote to select sellers over heterogenous middleware.

(b) Send a set of customised special offer messages to a respective set of customers (note the multiple messages). The two variables input to the scatter task are a list of channels and a list of messages.

Figure 6.41: Message multicasting.

6.3.8 Batch Filtering Patterns

The batch filtering patterns are listed with a reference to a YAWiL example in Table 6.12.
(a) Storing a batch of messages in process.

(b) Sending a batch of messages.

(c) Receiving a batch of messages.

(d) A minimum of ten transactions must occur before performing an audit.

(e) A minimum of three reviewer acceptances before allocating reviews to a paper.

(f) An alarm message from each of four monitors indicates a likely fibre-cut in a telecommunications provider.

Figure 6.42: Batch messaging.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Criteria Batch Filtering</td>
<td>BatFil42</td>
<td>Figure 6.43(a)</td>
</tr>
<tr>
<td>Message-Message Comparison Filter</td>
<td>BatFil43</td>
<td>Figure 6.43(b)</td>
</tr>
<tr>
<td>a-Priori Runtime Knowledge Filtering</td>
<td>BatFil44</td>
<td>Figure 6.43(c)</td>
</tr>
<tr>
<td>Better Messages</td>
<td>BatFil45</td>
<td>Figure 6.43(d)</td>
</tr>
<tr>
<td>Message Sort</td>
<td>BatFil46</td>
<td>Figure 6.43(d)</td>
</tr>
<tr>
<td>LIFO/FIFO Comparison Filter</td>
<td>BatFil47</td>
<td>Figure 6.43(e)</td>
</tr>
<tr>
<td>Batch Time-Schedule Comparison Filter</td>
<td>BatFil48</td>
<td>Figure 6.44(a)</td>
</tr>
<tr>
<td>Batch Frequency Spike Filter</td>
<td>BatFil49</td>
<td>Figure 6.45</td>
</tr>
</tbody>
</table>

Table 6.12: Batch filtering patterns.

6.3.9 Transformation Patterns

The transformation patterns are listed with a reference to a YAWiL example.
(a) Only process and review job applications for a given position if they meet the minimal eligibility criteria.

(b) Launch a customer loyalty program for any customer who applies for a set of loans totaling one million dollars (or more) in value. The Property Query Receive task uses a correlated SQL sub-query to perform aggregate operations over message properties Customer and Amount.

(c) All the purchase orders from the same customer need to be invoiced together each month.

(d) Accept the three better offers from a set of quotations and sort them by price.

(e) Accept the first thirty five customer applications for a sixty percent discount hotel guest package.

Figure 6.43: Batch filtering.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Message &lt;-&gt; Wire Message</td>
<td>Trans50</td>
<td>Figure 6.46 (YAWL Data Perspective (XSLT Transforms)).</td>
</tr>
<tr>
<td>Data Transformations</td>
<td>Trans51</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.13: Transformation patterns.

6.3.10 Discovery Patterns

The discovery patterns are listed with a reference to a YAWiL example.
(a) Begin processing interbank transactions at the end of each business day.

(b) Update stock prices if the message is less than ten seconds old.

Figure 6.44: More batch filtering.

Figure 6.45: Handle frequency spike: If more than one hundred alarms arrive within thirty minutes begin an alarm spike handling flow.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>#</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Registries</td>
<td>Disc52</td>
<td>Not supported - out of scope</td>
</tr>
<tr>
<td>Channel Mobility</td>
<td>Disc53</td>
<td>Figure 6.32</td>
</tr>
<tr>
<td>Process Mobility</td>
<td>Disc54</td>
<td>Not supported – out of scope</td>
</tr>
</tbody>
</table>

Table 6.14: Discovery patterns.

6.4 Implementation

Most of the extensions in YAWiL basically take advantage of YAWL’s innate extensibility. For instance, tasks for conveying messages in and out of processes are simply custom services in YAWL [4]. The implementation of YAWiL relies on the match-making service provided by JCoupling (see Chapter 4) as its foundation. It also incorporates fundamental ideas concerning coupling (see Chapter 3).
Figure 6.46: Transformation of process messages into wire-messages is automatically performed by JCoupling. An XML message in process A is forwarded by the send task onto the channel and through to the JCoupling Bridge. The JMS Adapter inside JCoupling injects the XML into a JMS object automatically.

### 6.4.1 Architecture

Figure 6.47 illustrates the as-built architecture of the implementation. JCoupling-2 encompasses several extensions to JCoupling. There is a JCoupling-Process Bridge (a custom YAWL Service) that integrates the YAWL engine to JCoupling. The JCoupling-Process Bridge translates work-item requests from the YAWL engine into requests to receive/send messages via the JCoupling service. The “Adapter Layer” provides an API that supports integration over heterogenous forms of middleware. The “JMS Adapter” links a JMS Server to JCoupling (and in turn YAWL). Another workflow engine is shown behind the JMS server to illustrate a JMS-aware workflow system and its potential integration points to the JCoupling 2 architecture. The “Controller” component is responsible for storing receive requests into the “Filters” repository. The controller continuously polls all registered channels for new messages arriving. New messages are copied down into the “Message” repository and message fingerprints are added to the “Fingerprints” repository. A message fingerprint is essentially an n-tuple built by applying properties to a message arriving on a channel into JCoupling. Most of the remaining implementation work requires enhancements to
the YAWL engine and the JCoupling-Process Bridge. There is also work to be done enhancing the YAWL process editor.

Figure 6.47: Architecture of the JCoupling 2 Implementation. This diagram was authored by J. C. Kuhr as part of the collaboration over the JCoupling-2 implementation (2009).

In Figure 6.47 the YAWL engine connects to JCoupling via a custom YAWL service called the JCoupling-Process Bridge. The JCoupling-Process Bridge provides the connectivity between the communication tasks in YAWL.
and the interaction/match-making services offered by JCoupling-2. The main components of JCoupling-2 are:

**Adapter Layer** The Adapter Layer provides a way for integrating adapters to heterogenous forms of middleware and workflow systems.

**Persistence Layer** The Persistence Layer stores Messages, Channels, Properties and active Filters.

**Filter Layer** The Filter Layer executes properties over incoming messages and stores the property values (*message fingerprints*) into special tables. The Filter Layer also executes active Filters over the *fingerprint* tables to find matching messages.

**Administration Layer** The Administration Layer allows users to add/delete/modify Channels and Properties.

The task of implementing YAWL was shared with a team of professional developers from Gecko (http://www.gecko.de) – based in Rostock, Germany. An initial prototype, able to perform match-making, was provided to Gecko. It was able to extract properties from messages and write them to database tables for match-making. Before Gecko took the development on it was not able to perform the *fully rounded* functionality that would be required. A specification of the static and dynamic aspects of the match-making service, along with the initial prototype, was provided to Gecko during a visit to their office. Gecko introduced the ability to bridge to a YAWL engine, and support for the creation/deletion (administration) of filters, and channels, integration with a production database and a high-speed, custom object-relational access layer. Their investment into this project was borne out of a smart hospital vision. They aim to integrate hospital information systems with proximity sensors and humans (medical staff). Some details about the smart hospital process are discussed in Section 6.4.2.

This implementation, at the time of writing, does not offer complete support for the YAWL language. As presented in Tables 6.15 and 6.16, the implementation supports twenty-six out of the fifty-four patterns. There is partial support for eight patterns, meaning that the intent of the pattern, while achievable, requires more than a little modelling effort. Twenty patterns are not currently supported by the implementation.

As can be seen from Tables 6.15 and 6.16 the implementation is not able to perform non-blocking communication. With some enhancements to the YAWL engine this could be supported.\(^9\) This is probably the most difficult

\(^9\)This would require the YAWL engine to support communication tasks being bound to YAWL Conditions (according the rules in Section 6.1.5).
<table>
<thead>
<tr>
<th>Patterns Category</th>
<th>Pattern</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupling</td>
<td>1 Blocking Send</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>2 Non-blocking Send</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>3 Blocking Receive</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>4 Non-blocking Receive</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>5 Time-Coupled</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>6 Time-Decoupled</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>7 Space-Coupled</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>8 Space-Decoupled</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>9 Space-Decoupled Topic</td>
<td>+</td>
</tr>
<tr>
<td>Message Consumption</td>
<td>10 Take</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>11 Copy</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>12 Clean</td>
<td>+</td>
</tr>
<tr>
<td>Request Response</td>
<td>13 Acknowledgement</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>14 Response</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>15 Throw Fault</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>16 Receive Response: blocking</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>17 Receive Response: non-blocking</td>
<td>–</td>
</tr>
<tr>
<td>Composed Interaction</td>
<td>18 Instance Channels</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>19 Request-response Correlation</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>20 Token-based Correlation</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>21 Property-based Conversations</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>22 Compound Property-Based Conversations</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>23 Conversation Nesting</td>
<td>–</td>
</tr>
<tr>
<td>Event-Based Process</td>
<td>24 Event-based Process Instance Creation</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>25 Event-Driven Task</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>26 And/Or/XOR Event-Driven Task</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>27 Event-Driven Choice</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>28 Unsolicited Events</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>29 Event-based Task Interruption</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>30 Timeout</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 6.15: Implementation coverage (+ = supports, +/- = partial support, – = not yet supported).

aspect to introduce. Support of non-blocking communication would unlock support for several of the event-based patterns currently not supported. It is also currently not supportive of conversation modelling and execution. The necessary programmatic hooks and handles are ready for these additional features.

6.4.2 e-Health Proposal

JCoupling-2, along with the extensions to YAWL, was proposed as a method for automating actions in the Clinic for Anaesthesiology and Intensive Care at
Rostock University Hospital [93]. The Rostock University Hospital Intensive Care Unit (ICU) had recently introduced a Patient Data Management System (PDMS) and was operating a central SAP I.S.H.Med Hospital Information System.

Increases in the data entry workload on doctors and nurses, along with higher customer expectations in terms of quality, were driving up operational costs and reducing the time hospital staff could dedicate to patient care. When a patient was moved from a general ward of the hospital into the ICU the staff were required to perform the following tasks manually:

- Update SAP Hospital Information System (HIS) regarding patient move from general ward to ICU.
- Transport the patient.
- Allocate ICU bed location to patient.

### Table 6.16: Implementation coverage (+ = supports, +/- = partial support, - = not yet supported).

<table>
<thead>
<tr>
<th>Patterns Category</th>
<th>Pattern</th>
<th>Implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filtering</td>
<td>31 Data-Message Comparison Filter</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>32 Time-Schedule Comparison Filter</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>33 Time-Timestamp Comparison Filter</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>34 Multicast</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>35 Scatter-Cast</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>36 Message Batch</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>37 Batch Send</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>38 Batch Receive</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>39 Minimum Batch Receive</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>40 Maximum Batch Receive</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>41 Multi-Channel Batch Receive</td>
<td>–</td>
</tr>
<tr>
<td>Batch Messaging</td>
<td>42 Criteria Batch Filtering</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>43 Message-Message Comparison Filter</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>44 a Priori Runtime Knowledge Filtering</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>45 Better Messages</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>46 Message Sort</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>47 LIFO/FIFO Comparison Filter</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>48 Batch Time-Schedule Comparison Filter</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>49 Batch Frequency Spike Filter</td>
<td>+</td>
</tr>
<tr>
<td>Batch Filtering</td>
<td>50 Process Message &lt;-&gt; Wire Message</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>51 Data Transformations</td>
<td>+</td>
</tr>
<tr>
<td>Transformation</td>
<td>52 Service Registries</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>53 Channel Mobility</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>54 Process Mobility</td>
<td>–</td>
</tr>
</tbody>
</table>
• Register patient and allocated bed with the ICU Patient Data Management System (PDMS).

• Brief the doctor and the operating anaesthetist of the new patient.

• The doctor to define any necessary prescriptions.

• The nurse to connect any monitoring devices and take a blood sample.

A solution that automated these basic tasks was required. The proposal outlined a system which: (1) automatically detected the moving patient and updated the HIS that the patient was relocated into the ICU, (2) automatically detected the patient being moved into a bed location and registered the patient into the ICU PDMS, (3) notified the nurse, doctor, and anaesthetist of the new patient and scheduled their respective duties.

Figure 6.48 describes a high-level architecture of a YAWL and JCoupling based-solution that would address the needs of the hospital and ICU staff. This solution requires some RFID Readers detecting the movement of a hospital bed into the ICU and detecting the eventual placement of the hospital bed. These RFID readers trigger JMS messages that get transmitted to JCoupling. JCoupling uses this information to correlate RFID events concerning the same bed and pass these over the YAWL Bridge – progressing a workflow for the patient in that bed. When required the YAWL process sends updates to the PDMS and HIS using JCoupling as a message bus. The patterns required to model a solution in this case study included: Pattern 1: Blocking Send, Pattern 3: Blocking Receive, Pattern 4: Non-blocking Receive, Pattern 6: Time-Decoupled, Pattern 8: Space-Decoupled, Pattern 10: Take, Pattern 28: Unsolicited Events and Pattern 31: Data-Message Comparison Filter.

At the time of writing the solution was not fully built. A proof of concept simulating the events generated by an RFID reader had been built. Work on the HL7 adapter (integrating the HIS and PDMS) would still be required to bring this system into production.

Figure 6.49 presents how JCoupling is going to support the proposed solution. It presents the central layers and design of JCoupling and the way they shall work with processes, ICU information systems and ICU hardware.

In Figure 6.49 the numbers track a typical interaction scenario: (1) A tag reader in the ICU will detect the RFID of the bed and (2) generate a JMS message on Channel Ch1. A JMS Adapter in the JCoupling Middleware Layer is repeatedly polling for new messages on JMS Channel Ch1. It consumes the JMS message containing the RFID (3). A component in the JCoupling Control Layer posts a request to the Persistence Layer to store the message
in a buffer (4). Next, the component in the Control Layer, upon finding a matching filter, makes a callback to the YAWL Bridge (5). The YAWL Bridge automatically correlates the callback message to the receive request received earlier from the YAWL Workflow Engine. The YAWL Bridge checks in the appropriate work-item. It contains the RFID in its data (6). The YAWL process responds to the bed movement notification. It composes and sends an HL7 message to channel Ch2 (7 – 10). JCoupling pushes the message through to an HL7 adapter (11) and ultimately onto SAP I.S.H.Med (12).

The idea of combining HIA systems, proximity sensors and a workflow systems in this manner led to a German patent application [92].

6.5 Conclusion

YAWiL is a language for creating process integrations from the simple, to the highly advanced. With YAWiL one can model the need for the interaction to be immediate or time-decoupled. YAWiL is one offering in a succession of process integration languages. However it does offer some distinguishing features. In the conceptual sense it overcomes BPELs main weakness. Abstract
BPEL and BPMN are simply too abstract and Concrete BPEL is too tightly coupled with middleware technology. None of these languages are able to model the essence of an immediate versus a time-decoupled interaction.

In terms of conceptual scope YAWiL combines elements of coupling/decoupling, event-handling, filtering, batch messaging, correlation, conversations into a cohesive integration language. These capabilities are able to be offered over an architecture that is able to offer core integration concepts up to the process modelling layer, while balancing the need to technology at sufficient levels of abstraction that technology agnostic solutions can be created. The most significant contributions YAWiL offer to the field of knowledge are its ability to include core aspects of any interaction style into the process integration model, and its ability to be executed. Nevertheless more validation of the language and implementation is required, particularly by process integration users.

The syntax of YAWiL formally defined in ORM and offers a precise semantics. It has been shown as a capable method for modelling the most complex integration scenarios put forward in the process integration patterns, and all but a few of those patterns are modelled with just a few graphic elements.

While the implementation is not yet complete the pilot is showing a high degree of stability at Rostock University Hospital. The architecture of JCoupling-2 and the YAWL Bridge is such that complete implementation could be built by extending (not rewriting) the current one.
Figure 6.49: A close up view of how the JCoupling architecture will support process integration in an ICU ward [93].
Chapter 7

Epilogue

This chapter concludes the thesis. It discusses the work presented and outlines how the proposals of this thesis satisfy the solution criteria introduced in Chapter 1. It also outlines further work and potential improvements.

7.1 Chapter Overview

Chapter 1 introduced the complexities and challenges of business process integration. It framed the complexity of business process integration in terms of availability, communication latency, platform heterogeneity, uncertainty, high volumes, and correlation of messages to the right process instance. It also discussed the criteria for assessing a solution to these challenges.

Chapter 2 presented an overview of related work in the fields of application integration and process integration.

Chapter 3 presented three dimensions of coupling for both uni-directional and bi-directional interactions. By providing a formal semantics for each of these dimensions the chapter demonstrated that selections in each of the coupling dimensions can be combined in any manner with no compromise over behaviour. Real world middleware can be defined in terms of their support for these coupling dimensions. Therefore these couplings dimensions are an effective modelling instrument for the middleware layer of a process integration.

Chapter 4 presented an architecture and implementation for filtering messages. The proposal was explored as a moderately abstract approach for supporting the correlation of messages to process instances. It was also explored in terms of its ability to support complex messaging scenarios. The chapter presented the ability of the architecture to handle large volumes of messages.

Chapter 5 presented a reasonably large number of process integration sce-
narios in the form of process integration patterns. This collection of patterns was proposed to be a measure of the sorts of scenarios that a language for process integration should be able to support.

Chapter 6 presented a language for process integration called YAWiL. The syntax of the language was defined as a highly graphical set of process modelling constructs that extend YAWL. A formal definition of the language syntax was also presented. The chapter showed each pattern from Chapter 5 as a vignette of a YAWiL process model. In order to validate the proposals of this thesis an implementation and a real-world pilot were described.

7.2 Solution Criteria

The solution criteria from Chapter 1.2.3 propose the qualities that a process integration solution should provide. The following section discusses the degree to which YAWiL meets these criteria.

Expressive

YAWiL is based on the process integration patterns and these patterns are based on state of the art systems, standards, and build on patterns of other research in the field. YAWiL, as such, should capture the relevant dynamic and static aspects of process integration:

- YAWiL can be expressed to integrate with manual technologies such as facsimile and manual remote processes. For instance a facsimile phone number would be modelled as a channel. The facsimile channel would need to have a Facsimile Middleware Adapter (see Figure 6.23). To extract property values out of facsimiles one could engineer a property for that purpose. It could use optical character recognition technology to read the facsimiles and could easily be plugged in.

- The way the interaction occurs over the underlying middleware - in terms of coupling - is highly visible in the process model.

- Using a Request-Response task the success/failure of an the interaction is made made clear to the process layer. The success of the request can be learned immediately upon receiving the response.

- Modelling synchronous/asynchronous interactions, publish-subscribe, request-response, fire and forget using YAWiL requires little effort.
• Combinations of messages, from totally different sources can be selected together based on business rules using the Multi Receive (see Figure 6.4(c)).

• Using non blocking receive tasks (see Section 6.1.5) it is possible to model a process only reacts to a certain combinations of events (e.g. see Figure 6.38).

• Using a time-coupled, thread-coupled channel it is possible to model a mission-critical integration. A time-decoupled channel can model a process integration that does not need to be available all the time.

• Conversations can be modelled using YAWiL (see Section 6.36). Conversations can group many interactions together, in a process. The interactions can occur over channels of heterogenous middleware.

• At the process modelling layer the coupling/decoupling capabilities of middleware can be seen graphically.

Precise
Process integration models, if they are to be executed on hardware, must be precise. YAWiL appears to be designed with sufficient precision and the constructs seem to be logically layered and internally consistent. The whole language syntax is formally defined. Furthermore the semantics in many aspects are formally defined. There is a formal semantics for coupling (see Chapter 3), and a formal semantics for message – process match making (see Chapter 4).

Conceptual
Conceptuality, is the quality of a language that helps users manage complexity. Conceptual languages selectively expose the relevant and hide the less relevant:

• A conceptual language would allow models to plug-in and out of alternative forms of middleware. In YAWiL each Channel must have a Middleware Adapter. By changing the middleware adapter for a channel the user can plug-into an alternative form of middleware, at design time. In YAWiL a channel is bound onto a middleware completely through the middleware adapter.
• A conceptual language ought to support the specification of a well-known interaction pattern (e.g. request-response etc.) without specifying a middleware technology. A publish-subscribe interaction can be specified by using a request-response channel that has not been bound to a middleware adapter. Such a model is abstract (see Section 6.1.8) and the interaction pattern has been specified.

• In YAWiL low level details (such as message object format, access locations of middleware and middleware logins) are performed by the middleware adapter. Thus they are abstracted away from the process integration model.

• YAWiL users can model message selection logic and conversation design without making demands on the partner process as to message format, process execution software or middleware. The JCoupling-2 implementation can perform all conversation and match-making services regardless of what specific software is being run at the partner process enterprise. Indeed the partner process does not even need to know about YAWiL or JCoupling-2.

Suitable

Suitability encapsulates the notion that the meta-model has strong alignment with the problem domain. A suitable language should allow process integration to be modelled in a direct manner [74,91].

• A process language should be able to effectively rule-in certain types of middleware and rule-out others; based on the match between the modelled intent and the respective middleware capabilities. Using YAWiL a channel can be declared to have any coupling (e.g. time-decoupled, publish-subscribe, request-response). Once defined this channel can only be bound onto middleware that supports its interaction pattern.

• It should be possible, with relatively little effort, to model the processing of large sets of messages. There are three types of multi-send tasks (Figs. 6.1(b), 6.1(c) and 6.1(d)) three types of multi-receive tasks (Figs. 6.4(b), 6.4(c) and 6.4(d)).

• It should require minimal effort to select messages based on their contents. Using YAWiL messages can be selectively consumed using one property and a one-line filter expression.
• The effort required to selectively consume messages should remain unaffected by middleware heterogeneity. In YAWiL the filter expression is middleware agnostic. The differences in middleware and message format are encapsulated to lower levels.

• A conversation, involving many related interactions with different partner processes, should be easy to model. Using YAWiL each interaction in a conversation can be modelled using one property, a simple filter expression and a binding to the relevant conversation (see Fig. 6.34).

Understandable

The longer term management of process integration models is important – given complexity issues and personnel changes for example. The models need to be understandable to both technical and business people.

• In order to share process models among users it is necessary that the models are concise as possible. The YAWiL language elements are frequently minimal and terse. The Request-Response task removes the need for correlation modelling – letting the middleware perform that purpose. Properties always simplify the specification of how to extract values out of messages for use in filters. Instance channels remove the need to worry about matching messages to process instances because each process instance is allocated a private channel.

• An easily understood language should have a highly graphic syntax. YAWiL has a graphic syntax.

• A process integration language should have the capability of presenting more than one participant process in a model. The Process Container construct in YAWiL is able to hold more than one cooperating process. See Fig. 6.14 for the conceptual syntax and Fig. 6.22 for a concrete example of two cooperating processes. This construct offers the foundations for a highly expressive, precise choreography language.

7.3 Further Work

To help evaluate the approach and test the implementation more data intensive case studies need to be conducted. This may involve extending or enhancing the smart hospital ICU scenario (see 6.4.2).

A graphical modelling tool needs to built. Furthermore there needs to be user acceptance studies performed on the language. These would need
to learn what elements of the language are preferred by end users and how appropriate they are to the problem they are trying to solve.

Considering that JCoupling-2 could be a message match making process-message match making service to thousands of process instances, computational overhead is a major concern. In computational terms there is a potentially large set of queries being continuously evaluated against a set of high volume tables. Nevertheless, there are some factors in favour of the approach we chose for JCoupling-2 that mitigate the table sizes and the size of the query set:

- The architecture is eager to match process instances with incoming messages. When a match is found, usually, the filter is removed and one (or more) message rows are deleted – reducing dataset size and the number of outstanding queries. Furthermore as the number of outstanding queries increases the likelihood of a matching message increases. Thus the number of rows in the message tables should be inversely proportional to the number of outstanding filters – helping the implementation to scale.

- It is likely that many process instances will have the same design. They will submit many structurally equivalent filters (with different literal values) concurrently. Prepared statements allow similar queries to be parsed only once. When they are re-executed only the literals change reducing load on the DBMS.

The JCoupling-2 implementation currently re-evaluates all queries when any new message arrives on any channel, or when a delta of time has elapsed (to re-evaluate any time-relative queries). The first optimisation would be to only re-evaluate a select set of the queries each time a new message arrives.

We could also optimise any select-project query, that does not contain a time-relative expression and/or a join. The strategy relies on the fact that all existing message/s in the channel/s have already been evaluated against the query and have produced no results. Therefore if the query is over only one channel and contains no join, then it can be optimized by only applying the query over the newly arrived message. If the query is over more than one channel an optimisation is still possible if we apply the query over the join of the tuple that represents the newly arrived message and any other relations referred to by the query. Thus greatly reducing the size of the joined result, making the query more efficient. To implement this we could use database triggers that get added when a query contains a select-project-join. If the trigger produces a result then there would be a callback to the message engine, and the trigger, itself, would be withdrawn.
Bibliography


A. Barros, G. Decker, M. Dumas, and F. Weber. Correlation Patterns in Service-Oriented Architectures. In Proceedings of the 10th International Conference for Fundamental Approaches to Software Engineer-


224


