UNIVERSITE DE NICE SOPHIA-ANTIPOLIS

ECOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THESE

pour l'obtention du grade de

Docteur en Sciences

de l'Université de Nice Sophia-Antipolis

Mention :

présentée et soutenue par

Nadia CEREZO

CONCEPTUAL WORKFLOWS

Thèse dirigée par *Johan MONTAGNAT*

soutenue le 20/12/2013

Jury :

M.	Hugues	BENOIT-CATTIN	Professeur	Rapporteur
Mme.	Mireille	BLAY-FORNARINO	Professeur	Présidente
M.	Oscar	CORCHO	Associate Professor	Rapporteur
M.	Johan	MONTAGNAT	Directeur de recherche	Directeur
M.	Gabriele	PIERANTONI	Research Fellow	Examinateur

CONTENTS

List of Figures vi

Listings ix

Acknowledgements x

Notations xii

1 Introduction 1
- 1.1 Simulations . . . 1
- 1.2 Scientific Workflows . . . 2
- 1.3 Challenges . . . 3
- 1.4 Abstraction Levels . . . 4
- 1.5 Entanglement of Concerns . . . 6
- 1.6 Goals . . . 8

2 State of the Art 10
- 2.1 Scientific Workflow Models . . . 11
  - 2.1.1 Interface . . . 11
  - 2.1.2 Model . . . 12
    - 2.1.2.1 Graph Type . . . 12
    - 2.1.2.2 Node Type . . . 12
    - 2.1.2.3 Edge Type . . . 12
  - 2.1.3 Abstraction Level . . . 15
  - 2.1.4 Comparison Matrix . . . 15
  - 2.1.5 Discussion . . . 17
    - 2.1.5.1 System . . . 17
    - 2.1.5.2 Model . . . 17
    - 2.1.5.3 Abstraction Level . . . 17
- 2.2 Separation of Concerns . . . 18
  - 2.2.1 Paradigm . . . 18
  - 2.2.2 Main general approaches . . . 19
    - 2.2.2.1 Subject-Oriented Programming . . . 19
    - 2.2.2.2 Role-Oriented Programming . . . 19
    - 2.2.2.3 Aspect-Oriented Programming . . . 20
    - 2.2.2.4 Feature-Oriented Programming . . . 21
  - 2.2.3 Separation of Concerns in Workflows . . . 22
    - 2.2.3.1 Separation of Concerns in Scientific Workflows . . . 22
    - 2.2.3.2 Aspect-Oriented Workflows . . . 23

2.2.4 Discussion . . . 24
2.3 Model-Driven Engineering . . . 24
2.3.1 Paradigm . . . 24
2.3.2 Unified Modeling Language . . . 26
2.3.3 Model Transformations . . . 28
2.3.4 Discussion . . . 29
2.4 Knowledge Engineering . . . 29
2.4.1 Semantic Data Models . . . 30
2.4.1.1 Entity-Relationship Model . . . 30
2.4.1.2 IDEF1X . . . 31
2.4.2 Ontologies . . . 31
2.4.2.1 Types . . . 32
2.4.2.2 Languages . . . 32
2.4.3 Semantic Web . . . 32
2.4.3.1 Resource Description Framework . . . 34
2.4.3.2 RDF Schema . . . 36
2.4.3.3 SPARQL Protocol and RDF Query Language . . . 38
2.4.4 Discussion . . . 41

3 Conceptual Workflow Model 43
3.1 Conceptual Elements . . . 43
3.1.1 Conceptual Workflows . . . 43
3.1.2 Graphical Convention . . . 44
3.1.3 Encapsulation . . . 45
3.1.4 Conceptual Links . . . 47
3.2 Abstract Elements . . . 48
3.2.1 Activities . . . 49
3.2.2 Specialized Activities . . . 49
3.2.3 Links . . . 49
3.2.4 Iteration Strategies . . . 50
3.2.5 Graphical Convention . . . 51
3.3 Semantic Annotations . . . 51
3.3.1 Type . . . 52
3.3.2 Role . . . 52
3.3.3 Meaning . . . 52
3.3.4 Compatibility . . . 52
3.3.5 Graphical Convention . . . 54
3.4 Fragments . . . 54
3.4.1 Graphical Convention . . . 54
3.4.2 Variables . . . 55

4 Transformation Process 57
4.1 Mapping . . . 57
4.1.1 Mechanisms . . . 58
4.1.2 Classification . . . 58
4.2 Weaving . . . 59
4.2.1 Steps . . . 62
4.2.2 Fragment to SPARQL Conversion . . . 63

4.2.3 SPARQL query . . . 65
4.2.4 Conflicts . . . 66
4.2.5 Clean-up . . . 67
4.2.6 Binding . . . 68
4.2.6.1 Node-bound Weaving . . . 70
4.2.6.2 Link-bound Weaving . . . 70
4.2.7 Classification . . . 71
4.3 Tools . . . 72
4.3.1 Merging . . . 72
4.3.2 Erasing . . . 73
4.3.2.1 Conceptual Links . . . 73
4.3.2.2 Annotations . . . 74
4.3.2.3 Sub-workflows . . . 75
4.3.2.4 Example . . . 76
4.4 Discovery . . . 76
4.4.1 Process . . . 76
4.4.2 Matching . . . 77
4.4.2.1 Match Quality . . . 78
4.4.2.2 Matching Query . . . 80
4.4.3 Ranking . . . 81
4.4.3.1 Ranking Principles . . . 81
4.4.3.2 Scoring . . . 82
4.4.3.3 Ranking for Conceptual Inputs . . . 83
4.4.3.4 Ranking for Conceptual Outputs . . . 83
4.4.3.5 Ranking for Conceptual Functions . . . 84
4.5 Composition . . . 84
4.5.1 Link Suggestion . . . 86
4.5.2 Producer Suggestion . . . 88
4.5.3 Consumer Suggestion . . . 90
4.5.4 Converter Suggestion . . . 91
4.6 Conversion . . . 94
4.6.1 XML in a nutshell . . . 94
4.6.2 To GWENDIA (MOTEUR) . . . 95
4.6.2.1 Converting Inputs/Outputs . . . 96
4.6.2.2 Converting Activities . . . 96
4.6.2.3 Converting Links . . . 97
4.6.3 To t2flow (Taverna 2) . . . 99
4.6.3.1 Converting Inputs/Outputs . . . 100
4.6.3.2 Converting Activities . . . 100
4.6.3.3 Converting Links . . . 101
4.6.4 To IWIR (SHIWA) . . . 102
4.6.4.1 Converting Simple Chains . . . 102
4.6.4.2 Iteration strategies . . . 104
4.6.5 Classification . . . 106
4.6.6 Discussion . . . 107

5 Validation **109**
5.1 Prototype . . . 109
5.1.1 Architecture . . . 109
5.1.2 Features . . . 110
5.2 Virtual Imaging Platform . . . 111
5.2.1 OntoVIP . . . 112
5.2.2 Workflow Designer . . . 113
5.3 Conceptual Workflow Model . . . 114
5.3.1 VIP Simulators . . . 114
5.3.1.1 FIELD-II . . . 114
5.3.1.2 SIMRI . . . 117
5.3.1.3 SimuBloch . . . 118
5.3.1.4 Sindbad . . . 118
5.3.1.5 SORTEO . . . 120
5.3.2 Simulator Template . . . 122
5.3.3 Conceptual Workflows . . . 123
5.3.3.1 FIELD-II . . . 123
5.3.3.2 SIMRI . . . 124
5.3.3.3 SimuBloch . . . 125
5.3.3.4 Sindbad . . . 126
5.3.3.5 SORTEO . . . 128
5.3.4 Discussion . . . 129
5.4 Transformation Process . . . 130
5.4.1 VIP Fragments . . . 130
5.4.1.1 Simple Sub-workflows . . . 130
5.4.1.2 Two Steps Function . . . 130
5.4.1.3 Split and Merge . . . 131
5.4.2 Use Case . . . 132
5.4.3 Discovery and Weaving . . . 133
5.4.4 Composition . . . 137
5.4.5 Conversion . . . 139
5.4.6 Discussion . . . 142

6 Conclusion **144**

A Detailed Frameworks **147**
A.1 ASKALON/AGWL . . . 147
A.2 Galaxy . . . 149
A.3 GWES/GWorkflowDL . . . 150
A.4 Java CoG Kit/Karajan . . . 151
A.5 Kepler/MoML . . . 152
A.6 KNIME . . . 153
A.7 MOTEUR/GWENDIA . . . 154
A.8 Pegasus/DAX . . . 155
A.9 SHIWA/IWIR . . . 155
A.10 Swift . . . 156
A.11 Taverna/SCUFL . . . 156
A.12 Triana . . . 157

A.13 VisTrails 158
A.14 WINGS 159
A.15 WS-PGRADE 161

B Conceptual Workflow Meta-Model **163**

C Fragment to SPARQL Conversion Example **166**

D Two Converters Chain Query **168**

E Conversion to t2flow **170**

F Conversion to IWIR **176**

G OntoVIP URIs **179**

H License **181**

Glossary **183**

Bibliography **200**

LIST OF FIGURES

1.1 Scientific Workflow Abstraction Levels 4
1.2 Scientific Workflow Concerns and Lifecycle 7
1.3 Scientific Workflow Concerns Entanglement Example 7

2.1 Control Contructs in Data-driven Models 13
2.2 Data Flow in Control-driven Models 14
2.3 Feature Diagram Example 21
2.4 Model-Driven Architecture - Abstraction Levels 25
2.5 UML Class Diagram Graphical Convention (excerpt) 27
2.6 UML Class Diagram Example 27
2.7 Linked Open Data cloud diagram by Cyganiak R. and Jentzsch A. 33
2.8 Small triple sample 35
2.9 RDFS inference example 37
2.10 DBpedia HTML result screenshot 39
2.11 SPARQL CONSTRUCT example 40

3.1 Graphical Convention - Conceptual Elements 45
3.2 Image Spatial Alignment Process Example at Multiple Abstraction Levels . . . 45
3.3 Conceptual Workflow Composite Pattern 45
3.4 Conceptual Link Restriction 47
3.5 Data Links and Order Links Associations 50
3.6 Graphical Convention - Abstract Elements 51
3.7 Compatibility between Elements and Annotations 53
3.8 Graphical Convention - Annotations 54
3.9 Graphical Convention - Fragments 54
3.10 Fragment Example 55

4.1 Transformation Process 57
4.2 Mapping Process 58
4.3 Classification of the Mapping model transformation 59
4.4 Weaving Example - Base workflow 60
4.5 Weaving Example - Fragment 61
4.6 Weaving Example - Result 61
4.7 Weaving Process 62
4.8 Fragment SPARQL Query - Example 65
4.9 Fragment SPARQL Query - Conflicts 66
4.10 Fragment SPARQL Query - Conflicts Fixed 67
4.11 Fragment SPARQL Query - Final Result 68
4.12 Binding Example - Alignment Process 69

4.13 Binding Example - Unbound Fragment . . . 69
4.14 Binding Example - Unbound Weaving Result . . . 69
4.15 Binding Example - Node-bound Fragment . . . 70
4.16 Binding Example - Node-bound Weaving Result . . . 70
4.17 Binding Example - Link-bound Fragment . . . 71
4.18 Binding Example - Link-bound Weaving Result . . . 71
4.19 Classification of the Weaving model transformation . . . 71
4.20 Merging Example . . . 73
4.21 Erasing - Links . . . 73
4.22 Erasing - Links Bypass . . . 74
4.23 Erasing - Annotations . . . 74
4.24 Erasing - Sub-workflows . . . 75
4.25 Erasing - Link Constraints . . . 75
4.26 Erasing - Example . . . 76
4.27 Discovery Process . . . 77
4.28 VIP Ontology (excerpt) - Registration Processes Taxonomy . . . 79
4.29 Matching Query . . . 80
4.30 Composition Process . . . 86
4.31 $\mathtt{X} \rightarrow \mathtt{Y}$ Converter Search Query . . . 92
4.32 Conversion to GWENDIA - Basic Structure . . . 95
4.33 Conversion to GWENDIA - Inputs/Outputs Example . . . 96
4.34 Conversion to GWENDIA - Activities Example . . . 96
4.35 Conversion to GWENDIA - Links Example . . . 98
4.36 Conversion to t2flow - Basic Structure . . . 99
4.37 Conversion to t2flow - Inputs/Outputs Example . . . 100
4.38 Conversion to t2flow - Activities Example . . . 100
4.39 Conversion to t2flow - Links Example . . . 101
4.40 Conversion to IWIR - Simple Chains Example . . . 102
4.41 Converting Iteration Strategies to IWIR . . . 105
4.42 Classification of the Conversion model transformation . . . 106

5.1 Prototype Architecture . . . 110
5.2 OntoVIP Excerpt - Simulations . . . 112
5.3 OntoVIP Excerpt - Dataset Processing Ascendance . . . 113
5.4 Link to VIP Workflow Designer Screenshot . . . 113
5.5 VIP Workflow Designer Screenshot . . . 113
5.6 FIELD-II - Result Example . . . 115
5.7 FIELD-II - Abstract Workflow (MOTEUR screenshot) . . . 116
5.8 SIMRI - Result Example . . . 117
5.9 SIMRI - Abstract Workflow (MOTEUR screenshot) . . . 117
5.10 SimuBloch - Abstract Workflow (MOTEUR screenshot) . . . 118
5.12 Sindbad - Result Example . . . 118
5.11 Sindbad - Abstract Workflow (MOTEUR screenshot) . . . 119
5.13 SORTEO - Result Example . . . 120
5.14 SORTEO - Abstract Workflow (MOTEUR screenshot) . . . 121
5.15 VIP Simulator Template Conceptual Workflow . . . 122
5.16 FIELD-II Conceptual Workflow . . . 123
5.17 SIMRI Conceptual Workflow . . . 124

5.18 SimuBloch Conceptual Workflow . . . 125
5.19 Sindbad Conceptual Workflow . . . 127
5.20 SORTEO Conceptual Workflow . . . 128
5.21 Simple Sub-workflow Fragment Example . . . 130
5.22 PET 2 Steps Fragment . . . 131
5.23 Split and Merge Fragment - Link-bound . . . 131
5.24 Split and Merge Fragment - Node-bound . . . 132
5.25 Use Case High-level Conceptual Workflow . . . 132
5.26 Mapping Use Case - After Weaving PET 2 Steps . . . 133
5.27 Mapping Use Case - After Weaving SimuBloch . . . 134
5.28 Mapping Use Case - After Weaving (node-bound) Split and Merge . . . 135
5.29 Mapping Use Case - Fixed Split and Merge Steps . . . 136
5.30 Mapped Use Case . . . 136
5.31 Use Case Link Suggestions . . . 138
5.32 Use Case Intermediate Representation . . . 140
5.33 Use Case GWENDIA Conversion Result (MOTEUR screenshot) . . . 141

A.1 Sample ASKALON Workflow . . . 148
A.2 Sample Galaxy Workflow . . . 149
A.3 Galaxy Dummy Cycle Example . . . 150
A.4 Sample GWES Workflow . . . 151
A.5 Sample Kepler Workflow . . . 152
A.6 Sample KNIME workflow . . . 153
A.7 Sample MOTEUR Workflow . . . 154
A.8 Sample Taverna Workflow . . . 157
A.9 Sample Triana Workflow . . . 158
A.10 Sample VisTrails Workflow . . . 159
A.11 VisTrails Dummy Cycle Example . . . 159
A.12 Sample WINGS Workflow . . . 160
A.13 Sample WS-PGRADE Workflow . . . 161

B.1 Conceptual Workflow Meta-model - Conceptual Part . . . 163
B.2 Conceptual Workflow Meta-model - Abstract Part . . . 164
B.3 Conceptual Workflow Meta-model - Semantic Part . . . 164
B.4 Conceptual Workflow Meta-model . . . 165

D.1 $\mathtt{X} \rightarrow ? \rightarrow \mathtt{Y}$ Two Converters Chain Query . . . 169

LISTINGS

2.1 Turtle example . . . 35
2.2 N-Triples example . . . 35
2.3 RDF/XML example . . . 36
2.4 RDF/XML example (alternative) . . . 36
2.5 SPARQL SELECT example query . . . 39
2.6 SPARQL CONSTRUCT example query . . . 40
4.1 Fragment to SPARQL Conversion - Fragment Example (abbreviated) . . . 64
4.2 Fragment to SPARQL Conversion - Result Query (abbreviated) . . . 64
4.3 Matching Query (T = target type) . . . 81
4.4 Producer Search Query (T = target type) . . . 89
4.5 Consumer Search Query (T = target type) . . . 90
4.6 X → Y Converter Search Query . . . 91
4.7 Conversion to GWENDIA - Inputs/Outputs Example . . . 96
4.8 Conversion to GWENDIA - Activities Example . . . 97
4.9 Conversion to GWENDIA - Links Example . . . 98
4.10 Conversion to IWIR - Simple Chains Example . . . 103
C.1 Fragment to SPARQL Conversion - Fragment Example . . . 166
C.2 Fragment to SPARQL Conversion - Result Query . . . 167
D.1 X → ? → Y Two Converters Chain Query . . . 168
E.1 Conversion to t2flow - Inputs/Outputs Example . . . 170
E.2 Conversion to t2flow - Activities Example . . . 171
E.3 Conversion to t2flow - Links Example . . . 172
F.1 Conversion to IWIR - Flat Inputs Example . . . 176
F.2 Conversion to IWIR - Dot Example . . . 177
F.3 Conversion to IWIR - Cross Example . . . 178

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, **Johan Montagnat**, for your trust, your invaluable advice and your tireless work to help me succeed. Without you, I would have never overcome the beast that my PhD turned into. When you took me under your wing, you warned me that it would be hard - about that you were right - and that I would eventually come to find something amiss with you - about that you were wrong.

My next thanks go to **Mireille Blay-Fornarino** for treating me like an equal from day one and often seeing more worth in my work than even I would. Your advice was incredibly helpful and your trust was immensely humbling. Thank you for all the time you gave me.

Next I want to bow to the generosity of the other members of my jury. I want to thank **Gabriele Pierantoni** for being not just a colleague but a friend and for always saying it like it is; **Oscar Corcho** for showing so much interest in my work even though it pales in comparison to yours in many ways; and **Hugues Benoit-Cattin** for taking a bit of a leap of faith with me. I am truly grateful for your time and humbled by your trust.

I will never forget the help **Régine Saelens** and **Gilles Bernot** provided me either. You were there for me when I most needed it, the make-or-break moment where, if not for your kindness, I might have given up. Thank you so much.

Next up is my favorite coffee drinking partner, **Franck Berthelon**. You helped me keep my head clear and my feet on the ground. More than anyone else, you made me feel like I belonged, like I was just another PhD student. I can never thank you enough for that.

I now want to thank **Sabine Barrere** for the tremendous patience with which you showed me the administrative ropes. You made my life a lot easier and I dare hope I did not make yours too bothersome in return.

Thanks are also due for **Michel Buffa**, **Alban Gaignard**, **Franck Michel** and **Olivier Corby**. Your patience is superhuman and you helped me a lot more than you probably suspect. Thank you for always taking me seriously.

A bit further away from home, but just as close to heart, I want to thank **Tristan Glatard** for being an extremely tolerant and insightful colleague; **Bernard Gibaud** for challenging me to do a better work and giving me precious advice to that end; **Kitti Varga** for always lifting the mood up no matter the adversity (any project is lucky to have you); **Bob Bentley** for being such a good sport (how is that for a French?); **Silvia Delgado Olabarriaga** for talking to me as if we were the same age and of similar accomplishments; and the researchers I had the incredible chance to meet at WORKS'11 after reading their names so often in the litterature, especially **Ian Taylor** and **Ewa Deelman**, for being nowhere near as intimidating in person as I had made you up in my head.

I also want to thank **Guillaume Husson**, **Pavel Arapov**, **Mathieu Acher**, **Macha Da Costa Barros** and **Simon Urli** for being perfect office-mates. You belied all apprehensions I ever had about office life twenty times over. Anyone sharing an office with any of you is extremely lucky, whether they know it or not.

I did not have the pleasure of sharing an office with **Javier Rojas Balderrama**, **Christian Brel**, **Fanny Dufossé**, **Filip Krikava**, **Ketan Maheshwari**, **Romaric Pighetti** or **Tram Truong Huu** and that is a shame, for I am sure I would have been the luckier one for it. Thanks for being exemplary teammates.

I now want to turn my gratitude towards **Loula Fezoui** who supported me throughout in so many different ways, up to and including proof-reading just about everything. This thesis would not be anywhere near its current state if it were not for you.

Outside of work, there are too many people I feel grateful towards to list here. Plus, I am afraid I would lose some friends if I tried and inevitably failed to make an exhaustive list. Anyway, if you know me, you probably already know how I feel about you. If, for some reason, you do not, do ask. Come on. I do not bite. :)

And last but not least, I want to thank you for taking the time to read this, whoever you are. Please do take a look at the Glossary too. It's full of definitions and URLs and funny acronyms, I bet you'll like it. ;)

And now for something completely different...

It's not so much about gratitude than simple acknowledgement. Here's a list of softwares and websites used in the making of this document, besides those mentioned in it:

- **BibTeX** to generate the bibliography;
- **detexify** to find some of the LaTeX symbols;
- **FreeFormatter.com XML Formatter** to indent long XML documents;
- **Google Scholar** to look up most references;
- **GraphViz** to create some of the figures;
- **LaTeX** to layout the whole document;
- **MacOS X Preview** to crop PDFs;
- **makeglossaries** to generate the glossary;
- **Microsoft PowerPoint** to create most of the figures;
- **Protégé** to browse and tinker with ontologies;
- **subversion** to backup the LaTeX source code;
- **texmaker** and **TeXnicCenter** to edit the LaTeX source code; and
- **WordReference** to find or check French→English translations.

NOTATIONS

The following conventions are used throughout the book:

- "*Text written in italic between quotes*" is either a direct quote from another work or a term commonly found in the litterature.
- $Text\ written\ in\ italic\ outside\ quotes$ is either a latin expression, such as *i.e.*, or a reference to a mathematical or algorithmical element, such as a function.
- **Text written in boldface** is simply emphasized over the surrounding text.
- `Text written in Typewriter style` is something technical, such as a literal value, a specific program name or a specific variable.
- Text written in dark blue is either a reference to a specific numbered item (*e.g.* a section, figure, listing, etc) or a term defined in the glossary.
- **Text written in dark blue and boldface** is a glossary term emphasized because it is inherent to this work. Most such terms are part of our model and many others are the names of specific processes defined in this work.
- [Text written in green between brackets] is a reference whose details can be found in the bibliography. Outside brackets, green text is reserved for URLs.

> *Text written in rectangles such as this one is a direct quote...*
>
> ...AND THIS IS THE AUTHOR OR THE REFERENCE.

Those slightly different rectangles...

...are for equations.
And the text in boldface above is the title.

CHAPTER 1

INTRODUCTION

1.1 Simulations

Ever since the conception of computers, scientists have used them to handle part of their scientific experiments for many reasons, including:

- performing **complex computations** that would take too long and be too error-prone for humans to perform;
- manipulating **digital data** that is captured as-is or converted from analog sources and is then processed by computers; and
- implementing **virtual experiments** that are used to model reality in order to avoid the costs and risks of practical experiments, to try impractical conditions or to deduce knowledge from the model itself.

Scientific experiments that are partially or entirely carried out via computers are called simulations. In the field of life sciences, they are commonly referred to as in-silico experiments by contrast to other types of experiments, *e.g.* "*in-vivo*", "*in-vitro*" and "*in-situ*" experiments.

Complete simulations very rarely consist of a simple program. Even in cases where all computations are performed inside one executable, there is most often need for data management. Indeed, **data** must somehow be:

- **fetched** from where it was captured or generated;
- **prepared** so that it is compatible with the program(s) using it as input; and
- **visualized, transformed or mined** so as to provide results to the scientist and/or input data for further simulations.

To this day, scientists have chained the various programs composing their simulations manually or automated them through *ad-hoc* scripting and generic tools like GNU Make[1]. Both methods, manual composition and automation via generic tools and script languages, are poor fits for simulations, for the following reasons:

- the exploratory nature of scientific analysis induces **frequent reuse and repurposing**, which easily become tedious enough to warrant dedicated systems;
- many simulations handle such huge **data volume** that it is impractical to handle manually and hard to handle reliably via scripting; and

[1] GNU Make: `http://www.gnu.org/software/make/`

- **distributed resources** – *e.g.* databases, data streams, web services, computing grids and clouds – have become an integral part of most simulations and are rather hard to access: scientists find they need to become experts of distributed algorithms and web technologies in order to leverage the wealth of available distributed resources.

There is thus a practical and growing need for systems dedicated to the automation of simulations and usage of highly-distributed heterogeneous resources.

1.2 Scientific Workflows

The need to automate the use of highly-distributed heterogeneous resources was first answered in the corporate world – years before the issue was identified as such in the scientific community – by the concept of workflow:

> “ *Definition - Workflow*
>
> *The computerized facilitation or automation of a business process, in whole or part.*
>
> WORKFLOW MANAGEMENT COALITION (WFMC)

Applying that definition to scientific processes suggests that any form of automation of a simulation should qualify as a scientific workflow. In this work, however, we focus on systems built specifically to model, maintain and perform simulations, *i.e.* what is commonly called scientific workflow frameworks.

Scientific workflow frameworks often target different scientific communities and most of them were developed independently by different teams with different goals; frameworks are therefore plenty and varied (see Section 2.1.4 for an overview). No standard has emerged yet, in the field of scientific workflows, however there are noticeable trends. For instance, most frameworks are compatible with web services, allowing access to a significant part of the distributed resources available online, and/or provide end-users with drag-and-drop interfaces to compose scientific workflows graphically, making it easier for beginners to start composing workflows and for everyone to read the workflows authored by others.

One of the most quickly recognizable trends for anyone comparing scientific workflow frameworks is that the vast majority of scientific workflow models the frameworks rely on – implicitly or explicitly – are based on directed graphs. That very common choice presents many advantages:

- it is the most **straightforward** way to represent the order in which tasks must be performed and/or how data is to be transferred between tasks;
- it is a graphical representation that is universal enough to be **accessible** for beginners, who can start composing simulation tasks without having to learn a language syntax; and
- it is especially **legible** in that it clearly highlights, for the human reader, things about the process that might otherwise be quite hard to detect or evaluate, *e.g.* complexity, resource consumption, bottlenecks, loops and unreachable code.

For all those reasons, the vast majority of systems opted for directed graphs despite their disadvantages, most notably the lack of scalability: huge graphs are hard to layout, to read and to process. In order to have a basis of comparison and cater to most systems, we have chosen to focus on scientific workflow frameworks whose models are based on directed graphs – though in some cases the user interfaces might not expose that fact.

1.3 Challenges

The main goal of scientific workflows is, by definition, to automate simulations. Interestingly though, scientific workflows are increasingly used by scientists to formalize and share not only the results of their experiments, but also the scientific processes that produced them. As that usage spreads, so does the need for other features.

For instance, the need for **provenance** – to relate data with the workflow and parameters that produced it, so as to facilitate reuse and peer validation – sparked a series of four Provenance Challenges[2] which led to the definition of the Open Provenance Model (OPM) and recently the World Wide Web Consortium (W3C) standard PROV, which are widely regarded as standards for provenance modeling and sharing. Another good example of a hot topic is **preservation** – to run a scientific workflow, and thus run all of its components and access all the necessary data, years after its design and publication. That growing concern in the field is the main goal of the Wf4Ever project [Belhajjame 12].

The present work focuses on three key aspects of scientific workflows:

- **accessibility**, *i.e.* the ease with which domain scientists can read workflows created by other people or create their own;
- **reuse**, *i.e.* the ease with which users can use workflows created by other people for the same or different goals (a case referred to as **repurposing**); and
- **comparison**, *i.e.* the ease with which someone can compare different scientific workflows or different scientific workflow frameworks.

Despite laudable efforts to make scientific workflows accessible and reusable, notably via easy-to-use Graphical User Interfaces (GUIs), it is widely recognized that most existing scientific workflow models remain complex to use for scientists who are not experts of distributed algorithms [Gil 07, McPhillips 09]. We argue that the main factors exacerbating that complexity are (i) the fairly **low level of abstraction** of most existing scientific workflow models and (ii) the **entanglement of different types of concerns** which adversely affects legibility.

Both factors stem directly from scientific workflow models. Indeed, they constrain the range of abstraction in which scientific workflows can be modeled and determine whether and how different types of concerns may be separated. Comparison is also hindered by the variety of models, absence of standard and difficulty of converting from a model to another. Hence the goal of this work:

> GOAL
> Create a new scientific workflow model to improve **accessibility**, ease **reuse** and allow **comparison**.

[2]Provenance Challenges: `http://twiki.ipaw.info/bin/view/Challenge/WebHome`

1.4 Abstraction Levels

Scientific workflows are used to formally model simulations so they can be performed automatically on computing infrastructures. There is an obvious gap between the level at which simulations are conceived (*i.e.* the scientific domain(s) of the end user) and that of enactment.

The vast majority of scientific workflow models lie between those two levels: more technical than the simulations they model, so that they can be executed, yet shielding the user from much of the complexity of distributed algorithms, web technologies and Distributed Computing Infrastructures (DCIs).

On the one hand, it is relatively common to call the level of abstraction of most scientific workflows the **Abstract Level** and that of enactment the **Concrete Level** [Yu 05]. On the other hand, to the best of our knowledge, there is no consensus as of yet for the name of the highest level of abstraction (*i.e.* the level of the simulations themselves): we call it **Conceptual Level** to contrast it with the **Abstract Level**, but it is sometimes called "*Abstract*", *e.g.* in [Garijo 11], and hence might confuse the unsuspecting reader.

Given the definitions we just established and since our goal is to improve accessibility, we need our scientific workflow model to be as close as possible to the scientific domain(s) of the end users and therefore to lie at the **Conceptual Level**.

METHOD (PART 1 OF 4)

Create a new scientific workflow model at the **Conceptual Level**.

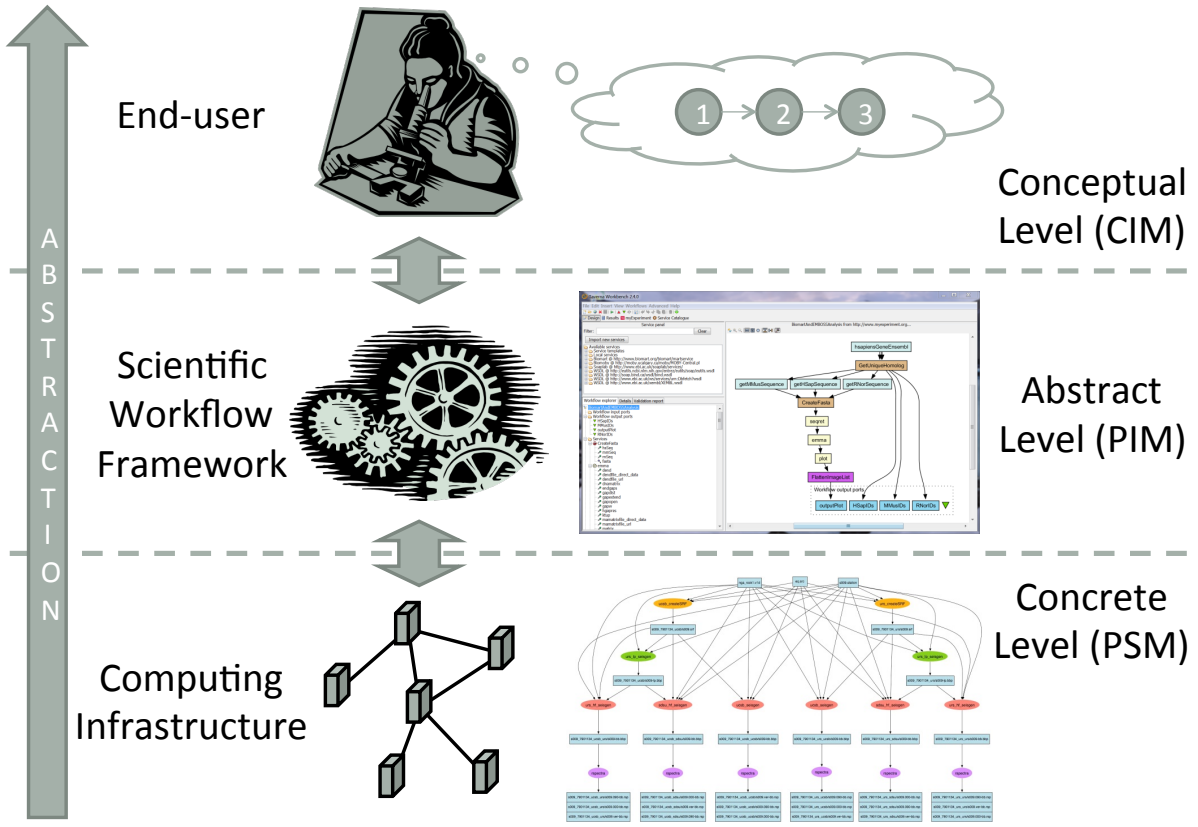


Figure 1.1: Scientific Workflow Abstraction Levels

Most scientific workflow frameworks invite users to work directly at the abstraction level of their scientific workflow model, bypassing the transformation from **Conceptual Level** to **Abstract Level** in the sense that the simulation itself is never formally defined and thus never actually transformed. However, the transformation from **Abstract Level** to **Concrete Level** can never be bypassed, since automation (and thus execution) is the main goal of scientific workflows. In most cases, that transformation is performed automatically by the scientific workflow framework.

We argue that the three levels of abstraction we just defined align very well with those of the Model Driven Architecture (MDA), as shown on Figure 1.1:

- The **Concrete Level** is that of the execution of scientific workflows by an enactor over a DCI. At this level, models are tightly-coupled with the infrastructure and are therefore referred to as Platform-Specific Models (PSMs) in the MDA.
- The **Abstract Level** is that of most scientific workflow models, ready to be automatically compiled or directly interpreted, but not entirely bound to specific resources and retaining some flexibility. Models at this level aim to be independent from computing infrastructures and are referred to as Platform-Independent Models (PIMs) in the MDA.
- The **Conceptual Level** is the one at which scientists conceive their scientific experiments in a vocabulary that is familiar to them. Conceptual models are referred to as Computation-Independent Models (CIMs) in the MDA, since they remain independent from how the system is or will be implemented. The distinction between CIM and PIM is much clearer in the MDA than it is in the field of scientific workflows, but few works have touched upon the transition from one to the other [Singh 10].

Theoretically, scientists can: (i) pick the scientific workflow framework which best suits their needs among a huge selection, (ii) design their simulations directly in the associated scientific workflow model and (iii) run their experiments on compatible DCIs.

However, in practice: (i) there are few surveys for users to compare existing systems and pick the most suitable [Yu 05, Taylor 07a, Barker 08, Curcin 08], (ii) designing at the **Abstract Level** requires technical know-how and knowledge about the target infrastructure, thus raising the entry barrier for scientists of all domains but that of scientific workflows and (iii) scientific workflow frameworks tend to be tied to their target DCIs enough to warrant multi-workflow systems interoperability projects like SHIWA [Krefting 11].

Hence the need not only to formalize simulations at the **Conceptual Level** but to develop a process to help users transform those representations down to the **Abstract Level**.

METHOD (PART 2 OF 4)

Develop a computer-assisted **Transformation Process** from the **Conceptual Level** to the **Abstract Level**.

There are three reasons why the **Transformation Process** must remain in our scientific workflow model as long as possible.

Firstly, without a closed-world assumption and a curated base of resources, the process cannot be fully automated. For one thing, it is impossible to guarantee that the resources needed to implement the user goals exist or can be identified as such. Since the **Transformation Process** requires user intervention, it would make sense not to jump from the scientific workflow model the simulation is designed in to a distinct model right at the start of the process.

Secondly, the frontier between **Conceptual Level** and **Abstract Level** is somewhat in the eye of the beholder: what passes for technicalities for one scientist might be core domain concerns for another one. It is therefore useful to cater to multiple levels of abstractions inside the same scientific workflow model, as has been recognized in the literature:

> *Workflow representations must accommodate scientific process descriptions at **multiple levels**. For instance, domain scientists might want a sophisticated graphical interface for composing relatively high-level scientific or mathematical steps, whereas the use of a workflow language and detailed specifications of data movement and job execution steps might concern computer scientists. To link these views and provide needed capabilities, **workflow representations must include rich descriptions that span abstraction levels and include models of how to map between them**. [...] Other important and necessary dimensions of abstraction are experiment-critical versus non-experiment-critical representations, where the former refers to scientific issues and the latter is more concerned with operational matters.*
>
> [GIL 07]

Thirdly, a conceptual scientific workflow model sufficiently independent from the lower abstract layer might serve as a basis of comparison between existing scientific workflow frameworks. One way to increase independence is to avoid picking a specific target abstract scientific workflow model and instead wait until the very last moment to delegate.

For those three reasons, the **Transformation Process** is divided in two distinct steps:

- **Mapping** is computer-assisted and transforms the scientific workflow from the **Conceptual Level** to an **Intermediate Representation** lying at the **Abstract Level**.
- **Conversion** is fully automated and converts the **Intermediate Representation** to a target language, so as to delegate to an existing scientific workflow framework.

And this split sparks the following need:

METHOD (PART 3 OF 4)

Extend the scientific workflow model with elements of the **Abstract Level** to model **Intermediate Representations**.

1.5 Entanglement of Concerns

Each phase of a scientific workflow lifecycle comes with its share of concerns, as summarized on Figure 1.2: (i) during **Design**, the simulation itself caters only to **domain concerns**, *i.e.* concerns pertaining to the scientific domain of the end-user; (ii) during **Implementation**, the scientific workflow must somehow fulfill all **technical concerns** in order to be executable; and (iii) during **Execution**, all manners of **non-functional concerns** become relevant, *e.g.* performance, reliability and provenance capture.

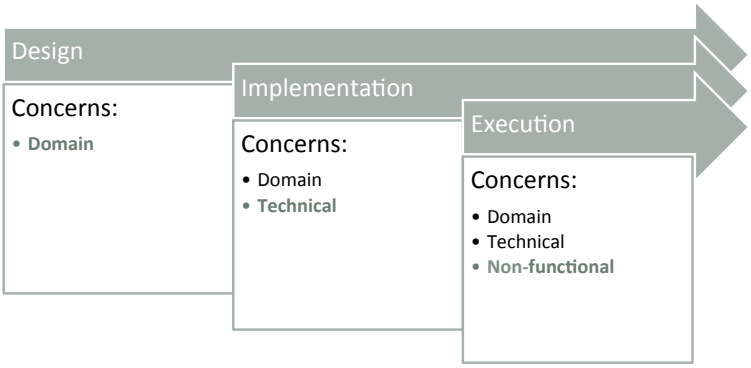


Figure 1.2: Scientific Workflow Concerns and Lifecycle

As an illustration of just how tangled those concerns can become, Figure 1.3 shows a Taverna [Missier 10a] workflow published by Paul Fisher on myExperiment[3] where we categorized each node as either **domain**, **technical** or **non-functional** based on its name.

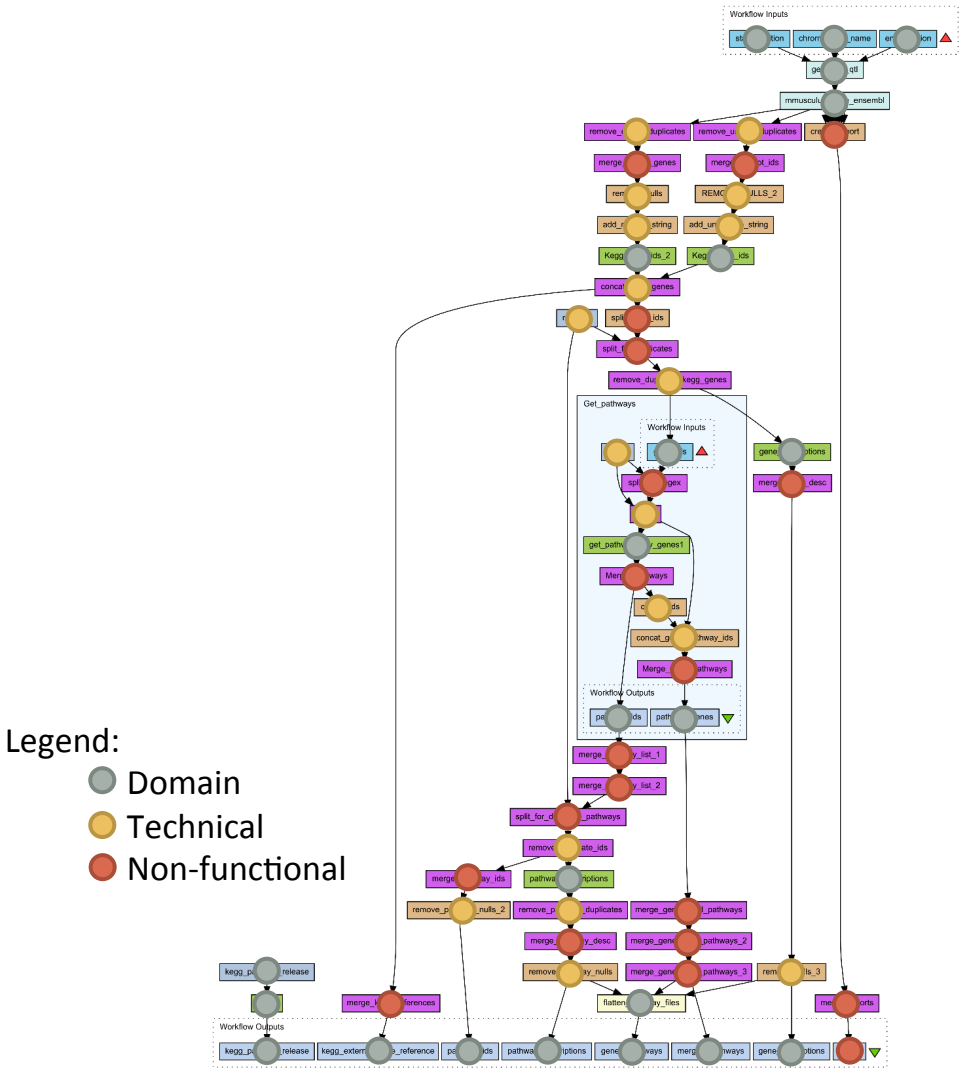


Figure 1.3: Scientific Workflow Concerns Entanglement Example

[3]Example workflow by Paul Fisher: `http://www.myexperiment.org/workflows/16.html`

It is fairly easy to see how indiscriminately mixing concern types can lead to much confusion when reading scientific workflows and much hardship when trying to maintain or repurpose them. The need to separate concerns in scientific workflows, not just inside one system but via multiple complementary ones, has already been expressed in the literature as such:

> *“ We also argue that in order to address all the important issues such as scalability, reliability, scheduling and monitoring, data management, collaboration, workflow provenance, and workflow evolution,* ***one system cannot fit all needs****. A structured infrastructure that separates the concerns of workflow specification, scheduling, execution etc, yet is organized on top of components that specialize on one or more of the areas would be more appropriate.*
>
> [ZHAO 08]

We thus argue that for a conceptual scientific workflow model to be truly effective, it must not only cater to the **Conceptual Level** of abstraction, but also allow proper Separation of Concerns (SoC), by allowing users to define concerns of different types separately from the base workflow they will be woven into.

METHOD (PART 4 OF 4)
Develop technologies to weave different types of concerns into scientific workflows.

1.6 Goals

Let us recap the goals of this work. In an effort to improve the accessibility of scientific workflows, we aim to:

1. create a scientific workflow model at the **Conceptual Level**;
2. extend the model with **Abstract Elements** to model **Intermediate Representations**;
3. develop technologies to assist the **Mapping** phase of the **Transformation Process**, from a high-level workflow to an **Intermediate Representation**;
4. develop technologies to automate the **Conversion** phase from an **Intermediate Representation** to a target scientific workflow model; and
5. develop technologies to weave concerns into scientific workflows to ensure proper SoC.

In Chapter 2, we present the state of the art of scientific workflow models as well as the domains our approach meets: model-driven engineering, separation of concerns and semantic knowledge engineering. We then present our scientific workflow model in Chapter 3 and the tools we provide - to transform **Conceptual Workflows** and delegate them to existing frameworks, including **Weaving** to improve SoC - in Chapter 4. Chapter 5 validates our contributions by applying them to real-life examples. Preliminary work was published in [Cerezo 11] and an earlier version of our contributions was published in [Cerezo 13].

CHAPTER SUMMARY

The **objectives** of this work are the design of a **scientific workflow model** encompassing **conceptual and abstract levels of abstraction** as well as the development of a **transformation process** supporting proper **Separation of Concerns**, in order to improve **accessibility**, ease **reuse** and allow **comparison** of scientific workflows.

CHAPTER 2

STATE OF THE ART

Long before they were used to model and automate simulations, workflows were defined, formalized and used in the industry for business processes. The research field of business workflows grew years before that of scientific workflows [Georgakopoulos 95] and has known many fundamental works and standardization initiatives such as the Workflow Patterns Initiative [Van Der Aalst 03].

The frontier between the two types of workflows, *i.e.* business workflows versus scientific workflows, is somewhat blurry and subjective. Nothing prevents a user from using a business workflow framework to model and perform a scientific experiment or a scientific workflow framework to capture and automate a business process. There are notably many efforts to use the *de facto* standard business workflow language Business Process Execution Language (BPEL) directly or adapt it for simulations [Emmerich 05, Akram 06, Slominski 07, Wassermann 07, Sonntag 13].

The differences between business workflows and scientific workflows pertain essentially to priorities and context, as detailed in [Barga 07] as well as in [Sonntag 10, Görlach 11], notably:

- The need for **security and privacy**, extremely important in a business context, is much less prevalent in the scientific community, where peer validation and collaboration are common goals that imply **sharing, reuse and repurposing**.

- The need for **integrity and reliability** is a central aspect of business services and thus a top priority for business workflows, but the exploratory nature of research makes **flexibility** a much greater priority for scientific workflows.

- Many business contexts require the fine-grained control and expressivity provided by **control-driven models**. However, scientific data is most often the first-class citizen of a simulation, which makes **data-driven models** and **hybrid models** better fits for most scientific workflows. In particular, data-driven and hybrid models can leverage data parallelism implicitly [Montagnat 09], which is crucial to many data- and compute-intensive simulations. The distinction between the three types of scientific workflow models is explained in Section 2.1.2.3.

- On the one hand, business workflow designers often face either (i) a lack of suitable candidate web services to perform a step in their process or (ii) a wealth of functionally close candidates which need to be differentiated through considerations of Quality of Service (QoS) and cost. On the other hand, scientific workflow designers often start modeling their simulations with the main services/programs already determined and generally find very few viable alternatives, since different candidates for a scientific process step often pertain to substantially different scientific approaches.

In this work, we focus on scientific workflows and scientific workflow models, analyzed and overviewed in Section 2.1, and not business workflows and their associated models. Elevating the abstraction level greatly improves accessibility but is not sufficient to ensure proper Separation of Concerns (SoC): that entails specific efforts which are somewhat transversal to abstraction levels, such as those described in Section 2.2. However, to be truly useful, high-level models and proper SoC must fit into a broader process to assist scientific workflow design. We chose to rely on an existing software development paradigm: Model Driven Engineering (MDE) described in Section 2.3. Section 2.4 outlines existing technologies to capture and leverage domain knowledge and know-how in order to assist the model transformations involved in our approach.

2.1 Scientific Workflow Models

In this overview of existing scientific workflow frameworks, we will analyze some of the most used systems along three dimensions: **Interface**, *i.e.* how users can create workflows and/or execute them; **Model**, *i.e.* the nature of the scientific workflow model used explicitly or implicitly; and **Abstraction Level**, *i.e.* where the system falls on the scale we defined in Section 1.4 (Conceptual, Abstract and Concrete).

2.1.1 Interface

There are plenty of ways for users to interact with scientific workflow frameworks. We will focus here on the following:

- **Application Programming Interfaces (APIs)** are protocols specifying how other programs may interact with the system. APIs let users generate, execute and/or monitor scientific workflows from inside other programs.
- **Command-line interfaces** let users execute scientific workflows directly from the command line. One of the advantages of doing so is that it greatly facilitates running scientific workflows on remote infrastructures.
- **Graphical User Interfaces (GUIs)** let users create and modify scientific workflows graphically. They often also provide a visual way to monitor their execution.
- **Dedicated file formats** were created and documented not just internally inside the framework development team, but for end-users to create and edit their scientific workflows in a format directly understandable by the framework.
- **Portals** target domain scientists and hide most of the complexity and technical details of scientific workflows. They even often hide the underlying scientific workflows entirely, focusing instead on input data and the scientific method in use.
- **Scripting languages** differ from dedicated file formats in that they are much more concise and are interpreted to generate the usually verbose executable format handled directly by the enactor.
- **Web services** are a particular kind of API meant to be used over the internet. We consider that they are available as an interface to a framework if said framework generates them automatically for the enactor and/or scientific workflow instances.

2.1.2 Model

Directed graphs lend themselves especially well to the modeling of processes, with nodes representing steps or locations and edges representing dependencies or transitions. Though there may be scientific workflow models wholly incompatible with directed graphs, most systems have either adopted them directly or adopted a compatible model.

We focus here on graph-based models so as to ensure some measure of comparability. We further distinguish the types of directed graphs used in each model based on the class of graphs used and the nature of the edges therein.

2.1.2.1 Graph Type

Directed Acyclic Graphs Plenty of scientific workflow frameworks (for instance Taverna [Missier 10a], *cf.* Section A.11) adopt Directed Acyclic Graphs (DAGs) as their core model for two main reasons: it is the closest representation to the way tasks are actually executed, no matter the target Distributed Computing Infrastructure (DCI); and it is very straightforward for data analysis pipelines, which represent a big part of scientific workflows.

Directed Cyclic Graphs Many scientific workflow frameworks (*e.g.* Kepler [Ludäscher 06], *cf.* Section A.5) opt instead for Directed Cyclic Graphs (DCGs) as their core model and thus allow loops to be modelled as such, instead of through constructs or dedicated activities.

Petri Nets Several systems (*e.g.* GWES [Neubauer 05], *cf.* Section A.3) have adopted petri nets as their core model, because it is a mathematical model designed specifically for distributed systems and it is already mastered by plenty of scientists who are eager to run simulations on DCIs.

2.1.2.2 Node Type

Nodes in graph-based scientific workflow models most often represent processing steps, *i.e.* either a simple invocation of a program, script or web service or a call to a sub-process or some abstraction thereof. Petri nets are an exception to that rule: half their nodes represent locations and the other half transitions between the locations.

2.1.2.3 Edge Type

Edges in graph-based scientific workflow models denote flow between the nodes: they specify how information and execution must be handled by the enactor. It is common practice to classify graph-based scientific workflow models into three categories according to the type of flow their edges represent [Shields 07, Deelman 09]:

- if all edges represent **data flow**, *i.e.* data transferring from a processor or location to the next, they are data-driven models;
- if all edges represent **control flow**, *i.e.* ordering constraints specifying that the target activity must run after the source activity, they are control-driven models; and
- if edges may represent **both flow types**, they are hybrid models.

Data-driven Models Edges in data-driven models (for instance Triana [Taylor 07b], *cf.* Section A.12) represent data flow between otherwise independent processing units. Perfect for digital processing pipelines and other streamlined processes, those models are, at first glance, the easiest to read and write.

However, their most useful feature - as far as scientific workflows are concerned - is implicit parallelism: on the one hand, processing units can run as soon as input data and the necessary computing resources are available, on the other hand, data sets can be split and dispatched to parallel computing resources automatically. As a result, those models shift the burden of planning asynchronous execution from the workflow designer to the enactor.

Unfortunately, elementary control constructs, such as conditionals and loops, are necessary for all but the simplest and most straightforward analysis pipelines. As a result, either data-driven models are extended with those constructs and become hybrid models or they emulate the corresponding behaviors, as illustrated on Figure 2.1:

- Straightforward ordering constraints can be implemented through dummy data tokens representing the control transmitted from the source to the target activity.
- More sophisticated control constructs must be implemented via dedicated activities handling them transparently from the viewpoint of the data-driven model.

Dedicated activities used to implement control structures tend to diminish the overall legibility of workflows in that knowledge of the workflow language itself is not enough: the reader has to identify every dedicated activity and decipher its meaning to understand the workflow logic. In practice, there is also a matter of portability with those activities, as they are often implemented via *ad-hoc* scripting and may well not run as-is in a different framework.

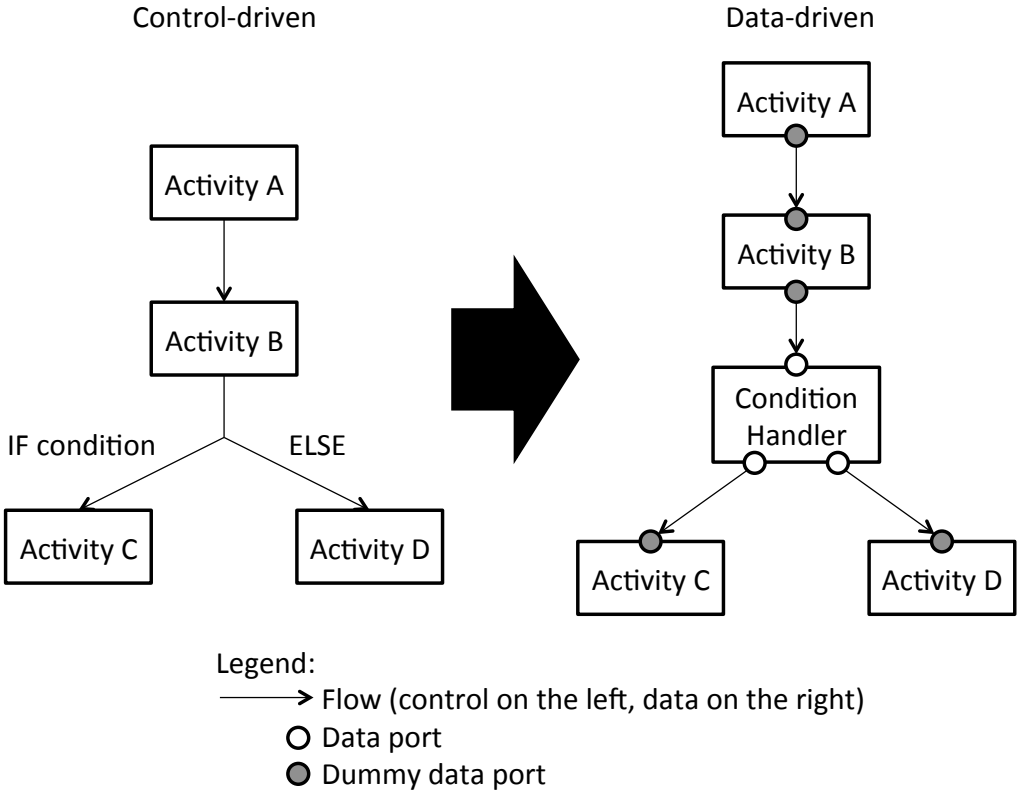


Figure 2.1: Control Contructs in Data-driven Models

Control-driven Models Control-driven models (*e.g.* ASKALON [Fahringer 07], *cf.* Section A.1) and Imperative Programming Languages (IPLs) follow the same fundamental principle: control is transferred from an operator to the next according to control constructs such as sequences, loops and conditionals. Because scientific workflows are meant to carry out simulations, unlike general-purpose IPLs, control-driven models:

- most often support graphical composition;
- aim at the best trade-off between expressivity and ease-of-use; and
- explicitly target parallel execution, notably by automatically handling the fine-grained elements thereof, such as messages and semaphores.

IPLs and control-driven models also differ by their respective abstraction levels: most IPLs are hardware-centric and deal with low-level operations such as variable assignment and memory allocation, but such tight coupling with the hardware would make scientific workflows unusable on heterogeneous DCIs. Therefore, control-driven scientific workflows deal with high-level operators, such as web services and grid jobs, and high-level control constructs that encapsulate complex transfer protocols between those operators.

Data is the first-class citizen in most simulations, yet it must be handled explicitly in control-driven models, which makes them less popular within the scientific community than within the business one. In control-driven models, data dependencies are modeled through intermediate activities that explicitly perform file transfers and handle synchronization from producer to consumer activity, as shown on Figure 2.2.

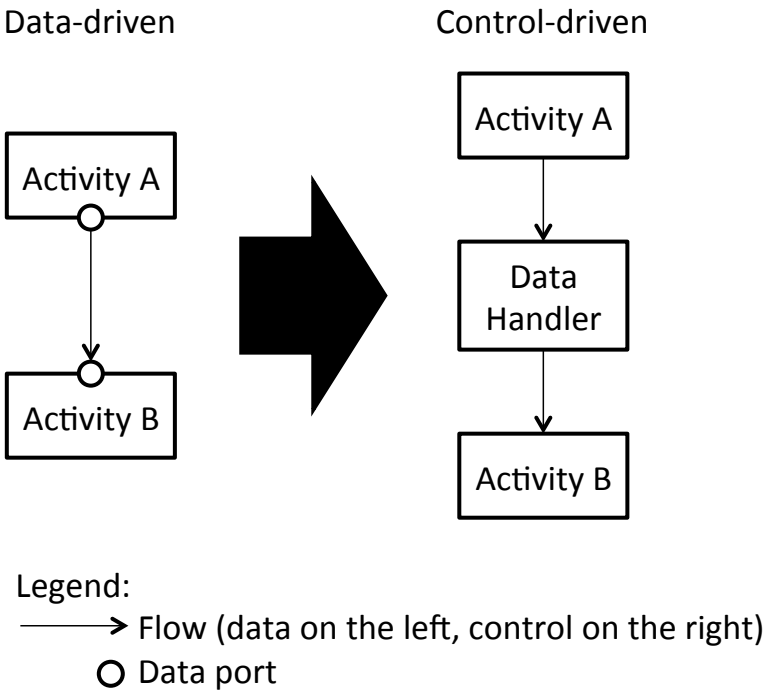


Figure 2.2: Data Flow in Control-driven Models

Hybrid Models The current tendency in the field of scientific workflows is to build hybrid models; most often data-driven models that are extended with select control constructs (*e.g.* MOTEUR [Glatard 08], *cf.* Section A.7).

> *“ It is clear that both control and data flow techniques are needed for scientific workflow languages. [...] Simple hybrid models with limited control constructs and support for data flow appear to stand the best chance of being interoperable with the most tools and frameworks but still contain enough functionality to be able to represent real scientific applications.*
>
> [DEELMAN 09]

Note that emulating control constructs in a data-driven model through dedicated activities, does not make the model itself hybrid. Indeed, the semantics of the model are unchanged and the control constructs are hidden from the enactor.

2.1.3 Abstraction Level

As stated in Section 1.4, most existing scientific workflow frameworks lie roughly at the **Abstract Level**. They are not however all at exactly the same level of abstraction.

Plenty of factors may contribute to elevate the abstraction level of a scientific workflow framework. We will focus on the following four criteria (the convention is that the level of abstraction is higher if the answers are positive than otherwise):

- **Annotations**: Can the scientific workflows and/or their components be annotated with **Semantic Annotations**? With curated keywords? With informal tags?
- **Composition**: Does the system automatically compose scientific workflows? Does it provide the user with suggestions of edges or nodes? Does it check existing edges for potential mismatches?
- **Flexibility**: Is there any structural flexibility in the scientific workflow model? Can the same scientific workflow instance represent or lead to (via generation/transformation) structurally different processes (*e.g.* a sequence of 3 tasks vs. 4 parallel tasks)? Is there flexibility in the data representation (*e.g.* multiple files can be represented by a single input parameter)?
- **Indirection**: Is there indirection between the specification of a task and the technical execution thereof? Can a given activity represent multiple web services? Multiple programs? Multiple processing units?

2.1.4 Comparison Matrix

Table 2.1 compares 15 of the most well-known scientific workflow frameworks along the dimensions described in the three previous sections. Details of each system can be found in Appendix A.

Table 2.1: Scientific Workflow Models - Comparison Matrix

Framework/Language	Interface	Model	Abstraction Level			
			Annotations	Composition	Flexibility	Indirection
ASKALON/AGWL	F, G	Control-driven DCG	◐	◇	◇	◇
Galaxy	G, P	Data-driven DCG	◆	◇	◇	◐
GWES/GWorkflowDL	C, F, G, P, W	Petri Net	◆	◇	◇	◆
Java CoG Kit/Karajan	C, F, G, S, W	Control-driven DCG	◇	◇	◇	◇
Kepler/MoML	C, G	Hybrid DCG	◆	◐	◐	◇
KNIME	A, G, P	Hybrid DAG	◐	◇	◇	◇
MOTEUR/GWENDIA	F, G, S, W	Hybrid DCG	◐	◐	◇	◐
Pegasus/DAX	A, C, F	Petri Net	◇	◇	◇	◐
SHIWA/IWIR	C, F	Control-driven DAG	◇	◇	◇	◇
Swift	S	Hybrid DCG	◇	◇	◐	◇
Taverna/SCUFL	C, G	Data-driven DAG	◐	◐	◇	◐
Triana	G	Data-driven DCG	◇	◇	◇	◇
VisTrails	G	Data-driven DCG	◆	◇	◇	◇
WINGS	G	Petri net	◆	◆	◇	◆
WS-PGRADE	G, P, W	Data-driven DAG	◇	◇	◇	◆

Interface: A: API, C: Command-line, F: Files (dedicated format), G: GUI, P: Portal, S: Scripting language, W: Web service

Abstraction Level Feature: ◆: Supported, ◐: Marginally present, ◇: Essentially absent, ◈: Third-party project(s)

2.1.5 Discussion

Though the previous section only describes some of the most well-known scientific workflow frameworks, it does give enough of an overview of the field to draw some conclusions.

2.1.5.1 System

There are enough scientific workflow frameworks available already and thus very little incentive to create yet another full-fledged framework, catering to scientific workflows from design to enactment. Instead, it seems more sensible to build either libraries (*e.g.* for provenance capture) for existing frameworks to rely on or stand-alone systems lying upstream (*e.g.* for design or sharing) or downstream (*e.g.* for enactment) from existing frameworks.

Since our aim is to elevate the abstraction level and assist the design of scientific workflows, we intend to build a stand-alone design system upstream from existing frameworks, *i.e.* a program focused solely on scientific workflows design and delegating management, enactment and so on to existing systems.

2.1.5.2 Model

Existing scientific workflow models vary widely and each solution presents advantages and flaws. For instance, DAGs are the most straightforward graph type and the easiest to check for errors, DCGs are more expressive but a little harder to check since they allow loops, endless loops included, and petri nets are already well-known to many scientific communities but are somewhat more complex to approach for complete beginners.

The trade-off between expressivity and accessibility is a complex one. Often, the most expressive models are the hardest to apprehend, but if a model is too simplistic, it will regularly require convoluted schemes. For instance, loops are commonly used in simulations and their implementation with data-driven DAGs, via dedicated activities, is uneasy and inconsistent (between different frameworks) enough to defeat the purpose of accessibility.

Our objective is to allow most scientists to model their simulations at the highest level of abstraction in as straightforward a way as possible. The best pick thus seems to be a nested directed cyclic graph which is mainly data-driven and hybridized with control constructs.

2.1.5.3 Abstraction Level

Though a lot of effort has been put into increasing accessibility to scientific workflow frameworks through easy-to-use interfaces and scientific portals which hide the underlying scientific workflows, the actual abstraction level of most existing scientific workflow models is pretty low.

We identified four features as key to elevating the abstraction level. Annotations and indirection are slowly becoming staples in the field and automated composition is certainly a hot topic. The new frontier is now the structural flexibility: even WINGS [Gil 11b], whose level of abstraction is clearly the highest in the field as of this writing, presents no such flexibility whatsoever.

That lack may be a legacy of business workflows. Indeed, while business processes evolve with time, they are nowhere near as variable as simulations, given the exploratory nature of science. It thus may seem reasonable to think of two structurally different business workflows as different and independent workflows, unlike scientific workflows where a given scientific protocol could and often is implemented in ways that significantly differ structurally.

For instance, the Virtual Imaging Platform (VIP), *cf.* Section 5.2, whose aim is to interoperate medical imaging models and simulators seminally included four simulators, and although the same general template holds true for all four simulators - and most simulators that have been added since or will be added in the future - the workflows, elements and structure, vary widely.

We argue that a scientific workflow model that would allow the modeling of all VIP simulators via a common scientific workflow, at the level of abstraction at which they are identical, would be beneficial in that it would ease the understanding of those applications as well as their comparison.

2.2 Separation of Concerns

Ensuring proper SoC is one of our goals, as stated in Section 1.6, and none of the systems analyzed in Section 2.1 achieve that particular goal. It is no surprise, given that structural flexibility is key to proper SoC and it is hardly supported in existing scientific workflow models.

2.2.1 Paradigm

The notion of SoC was first introduced by E.W. Dijkstra:

> *“Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one’s subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes have called “the **separation of concerns**”, which, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts, that I know of. This is what I mean by ”focusing one’s attention upon some aspect”: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.*
>
> [DIJKSTRA 82]

As the name and quote indicate, SoC is the name of a software design paradigm which can be described as such: programs often cater to wildly different (and sometimes contradictory) concerns and – in order to ease understanding, reuse, repurposing and modification – those concerns should be analyzed separately rather than mingled indiscriminately. This paradigm became more prominent over the last two decades [Hürsch 95].

2.2.2 Main general approaches

It is sometimes possible to separate concerns with the traditional techniques of encapsulation and modularization, but many typical concerns defy those techniques [Aldrich 00]. There is thus a need for software design approaches specifically dedicated to SoC. One of the earliest of such approaches focused on the notion of subjectivity and is subsequently called Subject-Oriented Programming (SOP).

2.2.2.1 Subject-Oriented Programming

At the basis of SOP is the notion that not every feature and function of an object is inherent to that object. In the now classical example of the tree, which to the best of our knowledge was introduced in the fundamental work [Harrison 93], it is easy to see that things like height and photosynthesis are intrinsic to the tree itself, whereas things like food value for birds and time to cut for lumberjacks are extrinsic.

An object can thus be built by composing "*subjects*": sets of fields and methods that are relevant to a specific application. This approach separates concerns of different applications, but it does so in a quite static and somewhat restrictive way.

In real life, the different ways an object is perceived based on beholder are indeed subjective views and the different ways objects behave and interact based on context are the "*roles*" they play. Hence the notion of Role-Oriented Programming (ROP).

2.2.2.2 Role-Oriented Programming

It is somewhat hard to track down the notion of ROP. It seems many concurrent works coined the term simultaneously [Belakhdar 96, Kristensen 96, Reenskaug 96].

Though ROP has been suggested in a more general context [Kristensen 96, Reenskaug 96, Demsky 02], it has been especially adopted in the field of multi-agent systems [Cabri 04]. Indeed, the notion of "*role*" might well be a more natural fit than that of function, when considering software agents, given that agent design focuses on autonomy and collaboration.

There are various ways to implement ROP. One of the most well-known frameworks is ROPE [Becht 99], which defines a domain-specific language and associated transformation process to design "*cooperation processes*".

Although ROP is an obvious good fit to model multi-agent systems, it does not handle all kinds of concerns equally well:

- **functional concerns** describe the system's basic functionality, *e.g.* what it is meant to do and how it does it;
- **non-functional concerns**, by contrast, cater to preoccupations and properties outside the basic functionality of the system, *e.g.* logging might be a very important part of a program's lifecycle, but, unless the program is a logging middleware, it is not a functional concern; and
- **cross-cutting concerns** are special non-functional concerns that significantly impact the structure of the process and/or are replicated in multiple points in the process and thus cannot be properly separated via modularization and encapsulation.

SOP and ROP divide objects along component lines that run parallel to traditional object classes and are as limited as traditional object-oriented programming when it comes to capturing cross-cutting concerns. For instance, whether a `Logger` is implemented as a class, subject or role does not change the complexity of tying it to every part of the system that must be logged. Hence the need for the more flexible notion of "*aspect*":

> *"The basis of the problem is that some kinds of behavior or functionality cross cut or are orthogonal to classes or components; these kinds of behavior appear in many components and are not easily modularized to a separate class. **Aspects** cut across or cross-cut the units of a systems functional decomposition (objects).*
>
> [KENDALL 99]

2.2.2.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) was conceived to tackle – and focuses on the issue – of cross-cutting concerns [Kiczales 97]. The aspect-oriented approach is to define cross-cutting concerns separately from the base process they are woven in and to weave them automatically. This is achieved by:

- the definition of a "*base process*" catering only to the functional aspects of the process;
- the declaration of "*join points*", *i.e.* precise locations in the base process where cross-cutting concerns might be woven;
- the specification of "*aspects*" composed of "*advice*", which define the additional behavior needed to fulfill the cross-cutting concern, and "*pointcuts*", which detect matching join points to determine where the aspect must be woven into the base process; and
- the use of an automated "*Weaver*" which will modify the base process with all the provided aspects.

The most well-known implementation of AOP is arguably AspectJ [Kiczales 01]. It is based on the Eclipse[1] framework and extends Java with aspects that can either use the typical combination of advice and pointcuts or directly and namely add methods, fields or interfaces to existing classes. It inspired similar extensions for a variety of languages, *e.g.* AspectC [Coady 01] for C, AspectC++ [Spinczyk 02] for C++, AspectS [Hirschfeld 03] for Smalltalk and many libraries[2] for the .NET framework. There are also language-independent AOP frameworks like Compose* [De Roo 08].

There are other Java libraries and frameworks for AOP in Java, notably JAC [Pawlak 01] which is a framework written in pure Java: it is less flexible than AspectJ in that programs must be built from the ground up following its logic and structure, but it is also more flexible in that aspects can be woven or unwoven at runtime.

[1] Eclipse: `http://www.eclipse.org`
[2] List of .NET AOP libraries: `http://www.bodden.de/tools/aop-dot-net`

Using AOP does not automatically imply using dedicated libraries and frameworks: AOP features have been incorporated into some languages like Perl[3] and are currently being incorporated into others like Ada[4].

One of the flaws of AOP is that the end result of weaving aspects into a base process tends to depend heavily on the order in which the aspects are woven. ADORE [Mosser 12] solves that problem by relying on logical foundations that allow interference detection and by mixing AOP principles with those of Feature-Oriented Programming (FOP).

2.2.2.4 Feature-Oriented Programming

Object-Oriented Programming, by focusing on the nature of objects and how they can be classified rather than on functionalities, tends to heavily limit feature composition. For instance, many Object-Oriented Programming languages do not allow multiple inheritance and thus something as simple as a square having all the features of both rectangles and rhombuses is hard to model. There is also a semantic problem in that saying that objects of type A share functionalities with objects of type B does not necessarily mean that they are similar in nature. Projects thus often end up with plenty of generic classes such as "*Sortable*" which do not describe the objects therein as much as what they can do or what can be done with them.

The paradigm of FOP aims to remedy that problem and is a bit reminiscent of iterative and incremental development: a program is made by stacking an ordered composition of "*features*" [Prehofer 93] which are formally defined and relatively small sets of functionalities. For that process to work out automatically, features are defined by how a program must be modified to acquire the associated functionalities. It is then a matter of applying all transformations sequentially, different composition orders generally resulting in different end programs.

Theoretically, the number of possible feature compositions grows exponentially with the number of available features. In practice though, not all feature compositions are viable or desirable. Inspired by industrial product lining, Software Product Lines (SPLs) analyze and define the exact set of viable and desirable feature compositions via "*feature models*" [Clements 01].

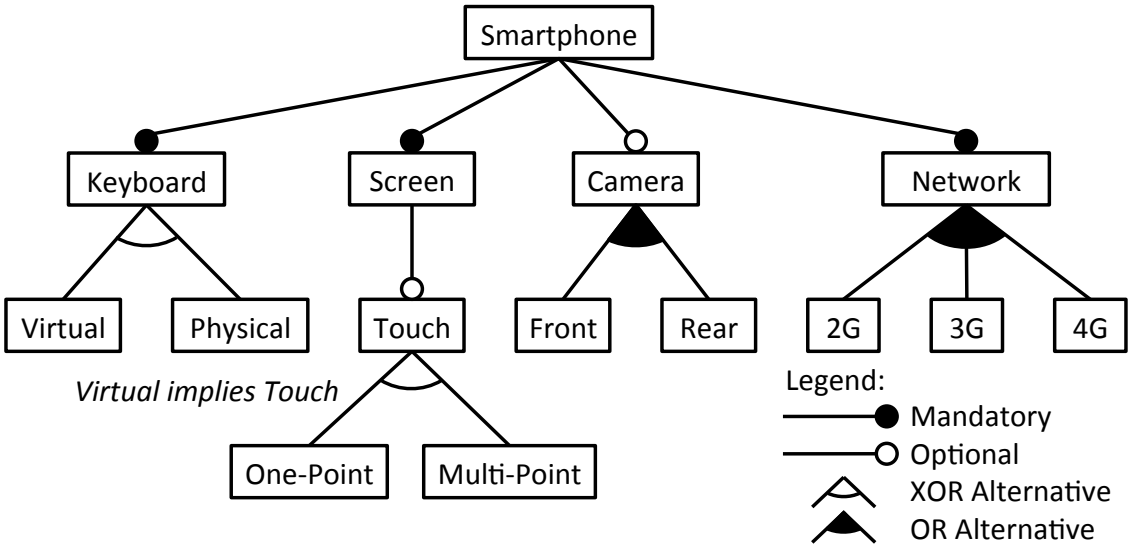


Figure 2.3: Feature Diagram Example

[3]Perl Aspect module: `http://search.cpan.org/dist/Aspect/lib/Aspect.pm`
[4]Ada 2012 Rationale: `http://www.ada-auth.org/standards/rationale12.html`

Feature models are essentially logic trees representing the full array of alternative sets of features for a system. They are represented by diagrams where the root of the tree, at the top, is the system (or sub-feature when a diagram is split) being described, where children of a node are sub-features of it and where the relationships between features are mostly indicated on the edges, with some constraints written in plain text, as shown on Figure 2.3.

FOMDD [Trujillo 07] is a software development paradigm which combines SPLs with MDE: programs are designed by composing features into models and transforming those models into executable artifacts. GenVoca [Batory 92] is a well-known model of SPL which defines programs as layers of unary functions. AHEAD [Batory 03] refines and extends GenVoca to handle multiple programs and representations. Program development in AHEAD is supported by the AHEAD Tool Suite [Batory 04].

Though extremely powerful, SPLs are limited by their closed-world assumption: the feature models most SPLs rely on not only define the precise set of possible features, but also the precise set of possible compositions thereof. However, there are frameworks, such as FAMILIAR [Acher 12b], that help overcome that limitation by managing the variability and composition of feature models, helping designers build and maintain feature models.

2.2.3 Separation of Concerns in Workflows

While, technically, workflows are a category of software, not all SoC approaches translate easily from the object-oriented paradigm to that of workflows.

2.2.3.1 Separation of Concerns in Scientific Workflows

To the best of our knowledge, Kepler [Ludäscher 06] is the first scientific workflow framework designed with SoC as a top priority [Bowers 05]. However, the concerns that the framework separates are not left for the workflow designer to choose. Unlike all other scientific workflow frameworks, Kepler separates "*communication*" and "*orchestration*": data-driven systems focus on the former, *i.e.* how data transits between activities; control-driven systems focus on the latter, *i.e.* how activities are scheduled; hybrid systems focus on neither and tangle both; and Kepler uses a specific "*Actor*" called the "*Director*" to specify how things are orchestrated and all edges in the graph represent pure data flow, disconnected from orchestration. That peculiar choice of concern separation is inherited from the underlying actor-oriented framework Ptolemy II[5].

A more recent work presented Chameleon [Bachmann 10]. Rather than a scientific workflow framework, it is a proposed architecture and a series of modular tools used to implement this architecture via integration environments. Like Kepler, Chameleon imposes a specific brand of SoC. It discriminates between three kinds of components in the scientific workflow framework: (i) data formats and data-related libraries; (ii) tools (*i.e.* activities), inputs and outputs; and (iii) the integration framework binding all other components.

Though the SoC provided by those two systems improves reusability, it does not at all let users determine which concerns to separate. Proper SoC cannot, therefore, be enforced in the scientific workflows themselves with those frameworks.

[5]Ptolemy II: `http://ptolemy.berkeley.edu/ptolemyII/`

At least two works have attempted to introduce proper SoC in scientific workflows by defining them with SPLs. [Ogasawara 09] takes advantage of SPLs to improve structural flexibility and to separate abstraction levels: an "*abstract*" workflow can be derived into various concrete workflows. [Acher 12a] uses SPLs to increase indirection and to assist workflow design: each choice made of an activity restricts the possibilities in the rest of the workflow, helping the design of "*coherent workflows*". Both approaches bring the power of SPLs to scientific workflows, but also its flaws: for those approaches to work, the variability of scientific workflows must be fully captured into feature models. That can be done in a specific use case or restricted domain, but would not work with an open-world hypothesis.

SoC is a much more advanced topic on the side of business workflows than it is on that of scientific workflows. As with the workflow frameworks themselves, it is quite natural to wonder whether state-of-the-art approaches to SoC can be directly transferred or adapted from one field to the other.

2.2.3.2 Aspect-Oriented Workflows

AOP is the most successful SoC approach in the field of business workflows, notably because:

- it focuses on cross-cutting concerns and most QoS concerns, which are often vital in business contexts, are cross-cutting;
- it can very easily separate concerns by domain in such a way that each expert only caters to the aspects of their own specialty, which fits the variety of experts commonly seen in the industry; and
- aspects can be used as a formal and flexible way to extend and maintain legacy programs and services without directly modifying them, a problem regularly faced by many companies.

There are many levels at which aspects may be introduced:

- at the enactor level, most generally a BPEL engine, aspects are used to adapt workflows dynamically and extend them with unforeseen features [Charfi 04, Courbis 05];
- at the service level, many works focus on the weaving of WS* standards, such as WS-Policy and WS-Security, rather than general-purpose aspects, with the added benefit of using well-established standards to express the non-functional cross-cutting concerns [Balligand 04, Ortiz 06]; and
- at the workflow level, some works focus on the generic Service Component Architecture instead of a given workflow language [Pessemier 08, Seinturier 09].

However, AOP is not as popular in the field of scientific workflows. It makes sense arguably, since the requirements of a scientist - or scientific team - performing simulations differ significantly from that of experts automating business processes.

On the one hand, aspects seem too restrictive for the extreme flexibility required by exploratory science, because "*pointcuts*" tend to be too tightly-coupled with the base process. For instance, if an aspect is described as coming "*before*", "*after*" or "*around*" a given service, it is not necessarily easy to reuse it in other ways or in other workflows. That somewhat restrictive quality of aspects is not an issue in most business contexts where fine-grained control is worth a lot more than flexibility.

On the other hand, aspects seem too sophisticated for the moderate needs of scientists: there is rarely a large team of vastly different experts collaborating on the design of one scientific workflow and cross-cutting concerns, while definitely present, are not as crucial for simulations as they are for business processes. It seems to us that AOP, though it definitely ensures proper SoC, is a bit of an overkill for scientific workflows and that is partly why it is hardly present in that field, especially compared to the field of business workflows.

2.2.4 Discussion

Our objective is to assist design and improve reuse of scientific workflows and to ensure proper SoC. We must thus be able not only to encapsulate sub-workflows into parent workflows, but also to divide and mix workflow parts, that must capture either a full scientific process (which can then be used as-is or encapsulated into a more complex scientific process), a piece of reusable know-how (such as a possible replacement for an activity) or a cross-cutting concern.

We call those parts **Fragments** (*cf.* Section 3.4), as they are close to "*workflow fragments*" defined in other works, most notably:

- they are sub-workflows or workflow parts annotated to facilitate their discovery, as in [Goderis 05, Wroe 07]; and
- they can be extracted from processes or created from scratch and are woven through model transformations during workflow design, as in [Schumm 11, Schumm 12].

However, the notion of "*workflow fragment*" is neither as precisely nor as formally defined as that of "*aspect*" in the litterature. The term is used in the field of business workflows in different contexts and with different definitions. For instance, in [Sadiq 01], a "*fragment*" is an element used to resolve "*pockets of flexibility*" at runtime, whereas in [Reese 06] it is a workflow part autonomous enough for different fragments to be handled by separate agents.

"*Fragment*" is also the name used in [Mosser 10] to qualify the components woven, to emphasize the differences with traditional aspects the method is inspired by, *i.e.* there is no distinction between a base process and aspects and the end result does not depend on the weaving order. Our notion of **Fragment** is comparatively less sophisticated and more flexible.

2.3 Model-Driven Engineering

A high-level scientific workflow model and a **Weaving** mechanism to merge **Fragments** would be of limited use unless they fit in a broader scientific workflow design process.

A scientific workflow is itself a model, but it is also a software. Scientific workflow design, the process we want to assist, falls under the purview of software engineering. It seems only logical to take inspiration from a model-based engineering method to structure our own design process.

2.3.1 Paradigm

MDE is a software development paradigm which revolves around and expands on the Model Driven Architecture (MDA) [Kent 02].

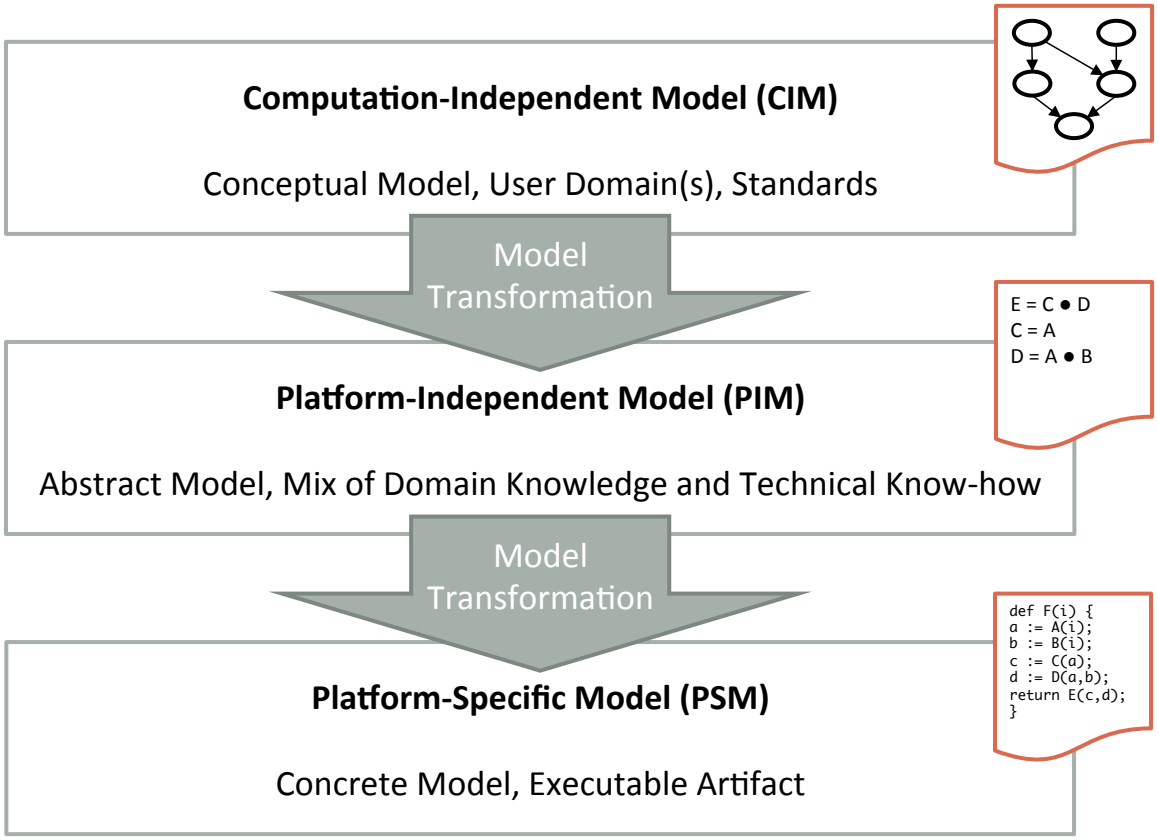


Figure 2.4: Model-Driven Architecture - Abstraction Levels

The MDA was devised (and the term copyrighted) by the Object Management Group (OMG) [Miller 03]. As illustrated on Figure 2.4, it defines three levels of abstraction for software models:

- Computation-Independent Models (CIMs) are at the highest level of abstraction, they are designed directly in the vocabulary of the user domain(s), according to their conventions and standards. They are called "*computation-independent*" because they focus on goals, instead of methods, *i.e.* on why the software should exist at all, what it is trying to achieve, rather than how it will be implemented to achieve those goals. For instance, a graph representing the architecture of a system is a CIM.

- Platform-Independent Models (PIMs) are at an intermediate level of abstraction, they mix domain knowledge and goals with technical know-how and methods. They are called "*platform-independent*" because they are not completely tied to a specific platform. For instance, the source code of a program written in a programming language, such as Caml, Fortran and Java, is a PIM. Some programming languages, however, lie at a substantially lower level of abstraction than most and tend to tie their source code to the target platform, most notably C.

- Platform-Specific Models (PSMs) are at the low level of abstraction of the target infrastructure and they can be executed/interpreted directly. They are called "*platform-specific*" because, more often than not, they cannot be executed on a different platform without modifications, which often turn out to be complex and have far-reaching impact. For instance, a compiled binary program is a PSM.

Based on that separation of levels of abstractions, the MDE principle is to build conceptual models in the problem domain(s) and use those models to generate executable artifacts more or less automatically. The benefits of such an approach include:

- **improved portability**, since conceptual models are independent from the infrastructure;
- **easier reuse**, since goals and methods are formally identified;
- **improved communication**, since models exist as a basis for communication between users of different domains, different projects or different teams; and
- **easier design**, since domain experts can focus on their own domain problems rather than on technical details.

The limitations pertain to three things: (i) **domain standardization**, which is anything but trivial; (ii) **duplication of work**, since the same kind of tools, whether for model design or model transformation, are re-developed by different teams, for different projects, in different domains or for different uses; and (iii) **automation of transformation**, which often requires complete specification of the problem space and thus a close-world assumption. So far, those limitations have proven strong enough to prevent widespread adoption of strict MDE.

However, many related ideas and methods are widely accepted and used. One of the most famous of such related items is arguably Unified Modeling Language (UML): a series of software meta-models ranging from CIMs to PIMs.

2.3.2 Unified Modeling Language

UML is an ISO standard created at the Rational Software company (later bought by IBM) and maintained by the OMG.

It formally defines 14 types of diagrams, each with its purpose, syntax and semantics. The "*Class Diagram*" is one of the most well-known and versatile of those diagrams. It is not restricted to software models and can be used to create meta-models of virtually anything.

We used the UML Class Diagram to model our own conceptual scientific workflow model and the elements used therein are introduced. For an exhaustive list of elements or for other diagrams, please refer to the OMG specification[6].

- The **Class** is the first-class citizen in a Class Diagram, as the name suggests, and it represents a type of objects, *e.g.* `Computer`, `GeometricalFigure` and `WebService`. It always has a name, but it can be further described by fields and methods (strict typing optional) that are either public (marked with a +), private (marked with a −), protected (marked with a #) or limited to the package (marked with a ~).
- The **Association** is the second-most important element of a Class Diagram, since it relates classes to each other. There are many types of associations, including:
  - the **Generalization**, which establishes inheritance: the source of the generalization is a sub-class, a specialization and the target is a super-class, a generalization, *e.g.* `Square` is a specialization of `Parallelogram`;
  - the **Composition**, which declares the source to be an integral part of the target that cannot belong to multiple objects simultaneously, *e.g.* an `Engine` is an integral part of a `Car`, but a given engine cannot belong to multiple cars;

[6]UML specification: `http://www.omg.org/spec/UML/Current`

  - the **Aggregation**, which is similar to the composition, but does not restrict the number of owners, *e.g.* a `Member` may be aggregated into distinct instances of `Club`.

- The **Multiplicity** restricts the number of associations that can exist simultaneously between an instance and others:
  - a number alone means that there must be exactly that many instances;
  - a wildcard `*` (or a `n` unknown) alone means that there can be any number of instances;
  - two numbers separated by two dots, *e.g.* `0..3`, means that the number of instances must be inside that range, limits included; and
  - a number and a wildcard `*` (or a `n` unknown) separated by two dots, *e.g.* `4..*`, means that the number of instances must be greater than or equal to the number. Consequently, `0..*` is the direct equivalent of `*`.

  An association often bears different multiplicities on each end. Indeed, a wheel can only belong to one car (or none), but a car better have four wheels. A composition from wheel to car might thus bear a `0..1` multiplicity on the side of the car and `4` on the side of the wheel.

- The **Package** indicates the boundaries of modules in the diagram, which may serve three purposes: (i) reflect structural, technical or conceptual boundaries; (ii) highlight parts of the diagram and name them; and/or (iii) identify sub-parts that are further detailed in other diagrams.

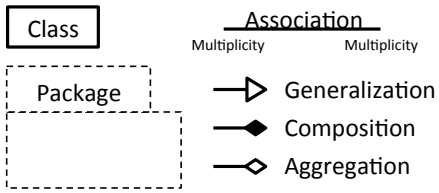


Figure 2.5: UML Class Diagram Graphical Convention (excerpt)

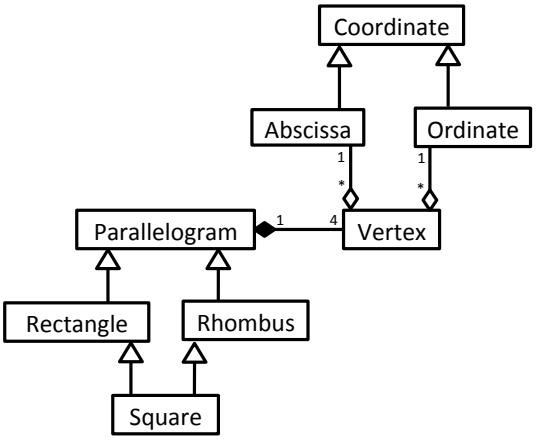


Figure 2.6: UML Class Diagram Example

Figure 2.5 illustrates the UML graphical convention for all aforementioned elements. To illustrate their use, let us consider the following example: squares are at the same time rectangles and rhombuses and all three are parallelograms. A parallelogram has four vertices. Each vertex has an abscissa and ordinate, which are both coordinates. Coordinates can be shared by any number of vertices. Figure 2.6 is a possible UML model for that small world of parallelograms.

Whether one uses UML or any other modeling language to design the CIMs and PIMs, without model transformations, models can only serve as a basis of communication, *e.g.* for developers to discuss specifications with clients. The MDE approach requires model transformations to implement a full-fledged software development paradigm, as intended.

2.3.3 Model Transformations

A model transformation is essentially a transformation process that takes a model as input and either modifies it or produces another model as output. Obviously, that is a loose definition that would include a plethora of things, from compilators to format converters.

In a sense, compilation of any programming language is indeed a form of model transformation, albeit at a rather low level of abstraction. Notably, the transformation rules are completely hidden from the user, unless he or she is an expert of the language, its specification and optimization techniques. With such black box model transformations, the regular user will focus on the meaning of his or her input model and its conformity to the demands of the model transformation, such as syntactic validity.

In the case of model transformation languages, however, the transformation rules are explicited and emphasized, taken as parameter alongside the input model. End users will not necessarily write the transformation rules themselves - they might instead reuse the rules of another user - but they will have a lot more access to them and will more likely than not have to pay attention, if only to pick the right set of rules to reuse. For instance, eXtensible Stylesheet Language Transformations (XSLT) is a transformation language anyone can use to define transformation rules from an eXtensible Markup Language (XML)-based language to another. Plenty of XSLT files can be found for a variety of well-known input and output languages. Specific needs, such as rather obscure input or output languages or unusual conversion choices, will require an ad hoc XSLT file.

In-between black box model transformations, with their implicit transformation rules, and transformation languages, there are plenty of hybrid systems, offering access to only part of the transformation rules. LaTeX[7] is a good example of such a system: a significant part of the transformation rules is hidden from users under the hood of the various compilators, but users can also specify their own transformation rules inside style files or directly inside the input model, *e.g.* using macros.

Of course, model transformations differ along not only the explicitness of transformation rules, but many other dimensions, including automation level. [Czarnecki 03] provides a detailed taxonomy of model transformations. We refer to it in Chapter 4 to classify the model transformations involved in our own approach. This taxonomy is formalized into feature models (see Section 2.2.2.4) detailing categories for:

- **Transformation Rules**: how they are expressed, how flexible they are and whether they are executable (or must be applied manually);
- **Rule Application Scoping**: whether it is possible to restrict the transformation to part of the source and/or part the target;

[7] LaTeX: `http://www.latex-project.org/`

- **Source-Target Relationship**: whether the transformation builds a new model or modifies the source in-place and whether updates are destructive (*i.e.* they may modify or remove elements);
- **Rule Application Strategy**: how it is determined which rules apply and which rule must be applied where;
- **Rule Scheduling**: whenever multiple rules must be applied, how the order in which they are is determined;
- **Rule Organization**: how rules relate to each other, whether they can be reused, combined and whether they are organized based on the target or source models;
- **Tracing**: whether there is explicit support for tracing elements between source and target models; and
- **Directionality**: whether a target model can be transformed back into the source.

Please refer to [Czarnecki 03] for the detailed sub-categories in each aforementioned category.

2.3.4 Discussion

Our approach fits right into MDE. Indeed, as explained in Section 1.4, the abstraction levels we outlined in the field of scientific workflows align with those of the MDA.

Moreover, the **Transformation Process**, detailed in Chapter 4, invites users to:

- design their simulation as a CIM, in our **Conceptual Workflow Model**, defined in Chapter 3, at the **Conceptual Level**; then
- transform that CIM into a PIM, called the **Intermediate Representation**, through **Mapping**, detailed in Section 4.1; and then
- transform that **Intermediate Representation**, through **Conversion**, into another PIM which can be converted into a PSM by an existing scientific workflow framework.

2.4 Knowledge Engineering

The model transformations involved in the aforementioned MDE approach progressively transform high-level conceptual models into executable artifacts. The starting conceptual models essentially represent domain knowledge about the problem at hand and the solution(s) adopted to solve it. The resulting executable artifacts, however, are most often made of technical know-how about how the adopted solution must be carried out. Somehow, model transformations must bridge the gap between those two fundamentally different yet interdependent worlds of domain knowledge and technical know-how.

The way to do it that least impacts the end-user is to bind specific domain knowledge to specific technical know-how, *e.g.* specify that a given high-level operation may be implemented by a given list of programs, generally either in extensive domain term definitions, such as a vocabulary of possible operations with lists of relevant programs, or in enriched technical specifications of executable artifacts and composites thereof, such as a repository of web services annotated with the operations they perform.

There is an obvious trade-off between flexibility and automation: with a closed-world assumption, *i.e.* with a limited and specified set of goals and methods involved, it is possible to guarantee that any high-level model created within the limits of the system may be automatically transformed into an executable artifact. However, it is not enough to restrict a system to a specific field of application to ensure full automation; the range of applications must be carefully restricted and curated as well or there cannot be any guarantee that relevant methods exist and can be matched to goals.

We think the closed-world assumption is too heavy a price to pay in the field of sciences because it is an especially evolutive environment where end users are often service providers in turn and because the ontologies modeling data and its processing are being developed in growing numbers and are themselves a moving target for now. Therefore, we do not want to restrict this work with a closed-world assumption; on the contrary, virtually any simulation could be modeled by **Conceptual Workflows**, defined in the following Chapter 3. In exchange for that flexibility, we forego any and all guarantees of automation: anything can be modeled at a high level of abstraction, but there might not be a method to operationalize it or, worse, there might be a method but no way to identify it as such.

To automatically detect whether an available operation is relevant to a given goal, its function and interface must be described in a processable way. Capturing domain knowledge about goals, capturing technical know-how about methods and inferring links between the two, all fall under the purview of the vast field of Knowledge Engineering research. The name seems to have been first coined by artificial intelligence experts Edward Albert Feigenbaum and Pamela McCorduck:

> “ *KE [**Knowledge Engineering**] is an engineering discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human expertise.*
>
> [FEIGENBAUM 83]

For a computer to handle and capitalize on human knowledge, that knowledge must first be represented in a way that computers can process: a digital model.

2.4.1 Semantic Data Models

Semantic data models are conceptual data models that not only describe data, but its meaning, notably how some parts relate to others. One of the earliest semantic data models, is also one of the most well-known and widely used: the Entity-Relationship model.

2.4.1.1 Entity-Relationship Model

The Entity-Relationship Model was devised by Peter Chen in 1976 [Chen 76] as an abstract way to describe databases, especially relational databases.

The model identifies three distinct types of concepts: **entities** are uniquely identifiable and may exist in isolation, *e.g.* students, teachers, classes; **relationships** relate entities to each other, *e.g.* a student attends a class and a teacher teaches a class; and **attributes** qualify specific entities, *e.g.* a student has a first name, last name and identification number.

When designing a relational database based on an Entity-Relationship Model, each entity is converted into its own table, each attribute becomes a column in that table and relationships are modeled based on their **cardinality**:

- a one-to-one cardinality suggests that one of the two entities could be designed as an attribute of the other, *e.g.* if each student has a mentor and each mentor is assigned to one student at a time, then the mentor can be an attribute of students;
- a one-to-n cardinality will translate to a "*foreign key*" (a column reproducing identification keys from another table) in the table which is on the n side of the relationship, *e.g.* teachers who work as advisors to groups of students will be identified as such by a foreign key in the student table; and
- an n-to-n cardinality will require a dedicated table to represent the relationship, with each row being a pair of foreign keys and related attributes, if any, *e.g.* the affectation of students to classes can be represented by a table with columns for a class, a student and a grade that student got in that class.

Because designing an Entity-Relationship Model is fairly easy, with a little experience, and transforming one into a relational database schema is extremely straightforward, that model has been and still is extremely popular.

It is however fairly skewed toward relational databases and thus is rarely the model of choice when trying to model semantic data in any other context.

2.4.1.2 IDEF1X

Based on the Entity-Relationship Model as well as relational theory at large, the US Air Force's Integrated Computer-Aided Manufacturing program developed a family of modeling language called the Integrated DEFinition (IDEF) methods[8]. IDEF1X [Kusiak 97] is one of those languages and is a fairly generic semantic data modeling language, aimed at integration of information systems.

On top of notions already present in the Entity-Relationship Model, IDEF1X adds:

- **domains**, *i.e.* set of data values that function as types for attributes;
- **categorization relationships**, *i.e.* a relationship defined semantically as a generalization, meaning one entity is a sub-type of the other; and
- **view levels**, which formally define different levels of abstractions and map them to distinct views on the same model.

Though semantics are more fleshed out in an IDEF1X model than in an Entity-Relationship Model, it is still a model dedicated to databases and that is apparent in the lack of instance to instance logic: everything is expressed at the level of classes, notably the affiliation of entities and attributes to domains.

2.4.2 Ontologies

Ontologies are semantic data models augmented with the notion of reasoning: they not only contain information about concepts and how those concepts relate to each other, but about how new information can be inferred from already stated information.

[8]IDEF methods: `http://www.idef.com/`

2.4.2.1 Types

There are many types of ontologies, built in different ways with different objectives.

A taxonomy is the most basic type of ontology: it is a classification of concepts, a hierarchy of classes. Folksonomies are a peculiar kind of taxonomy, obtained not through design by a handful of people, but by an entire community freely tagging content. However, they are generally flat vocabularies and, as such, barely qualify as taxonomies unless, for instance, hierarchical clustering is employed [Shepitsen 08]. [García-Silva 12] overviews and analyzes approaches to enrich folksonomies with semantic data.

At the very least, a taxonomy requires two relations: an affiliation relation, meaning that an instance belongs to a given class, and a generalization relation, meaning that all instances of a class are automatically instances of another, *e.g.* all instances of `Shark` are instances of `Fish`. Some taxonomies go well beyond those two relations. For instance, the foundational ontology WordNet [Fellbaum 10] specifies synonymy (words mean the same thing), antonymy (words mean the opposite of each other) and other more complex relations, notably meronymy (a word is composed of other words).

Foundational ontologies, such as DOLCE [Gangemi 02], also known as upper or top-level ontologies, are particularly generic and abstract: instead of targeting a specific domain, they attempt to unify multiple domains by identifying universal concepts and inference rules. Their biggest advantage is undoubtedly interoperability: they can be used to bridge the gap between different domains or different domain-specific ontologies. But there are obvious trade-offs between genericity and accessibility, on the one hand, since the vocabulary is often far removed from the one used in any specific domain; and between universality and expressivity, on the other hand, since universal inference rules are unlikely to fully capture the knowledge of any specific domain.

2.4.2.2 Languages

There are plenty of ontology languages, specifying how data and inference rules must be represented and thus constraining what inference rules can be expressed.

Some ontologies are based on first-order logic, notably the language of the Cyc foundational ontology, CycL [Lenat 91] and the Knowledge Interchange Format (KIF)[9] which evolved into the ISO Common Logic standard[10].

Others, like F-Logic, which stands for Frame Logic, try to establish logical foundations for object-oriented modeling [Kifer 95].

A family of ontology languages have been developed specifically for the web and are based on description logic, notably the DARPA Agent Markup Language (DAML)[11] and the Ontology Inference Layer (OIL) [Fensel 01], merged into DAML+OIL [McGuinness 02].

In 2002, the World Wide Web Consortium (W3C) picked up DAML+OIL and revised it into Web Ontology Language (OWL), which is now the standard ontology language for the Semantic Web.

2.4.3 Semantic Web

As the name suggests, the Semantic Web is an extension of the web with semantic technologies. It inherits advantages and flaws of both.

[9] KIF: `http://www.ksl.stanford.edu/knowledge-sharing/kif/`
[10] Common Logic standard: `http://iso-commonlogic.org/`
[11] DAML: `http://www.daml.org/about.html`

Like the web, the Semantic Web is an open and decentralized space, which makes it possible for anybody to participate freely, but also makes collecting and curating information an impossible utopia: by design, there is neither a way to enforce any convention beyond basic protocols, nor ever any certainty that one has access to all relevant data at any given time (data could be unreachable or in the process of being added).

Like all semantic technologies, the Semantic Web makes data and data-driven applications smarter, but at the heavy upfront cost of domain modeling. Ontology design is a craft that is not easily mastered and takes a lot of time. Annotating data for the first time, rather than converting from one semantic data model to another, is extremely error-prone if automated and extremely tedious if done manually.

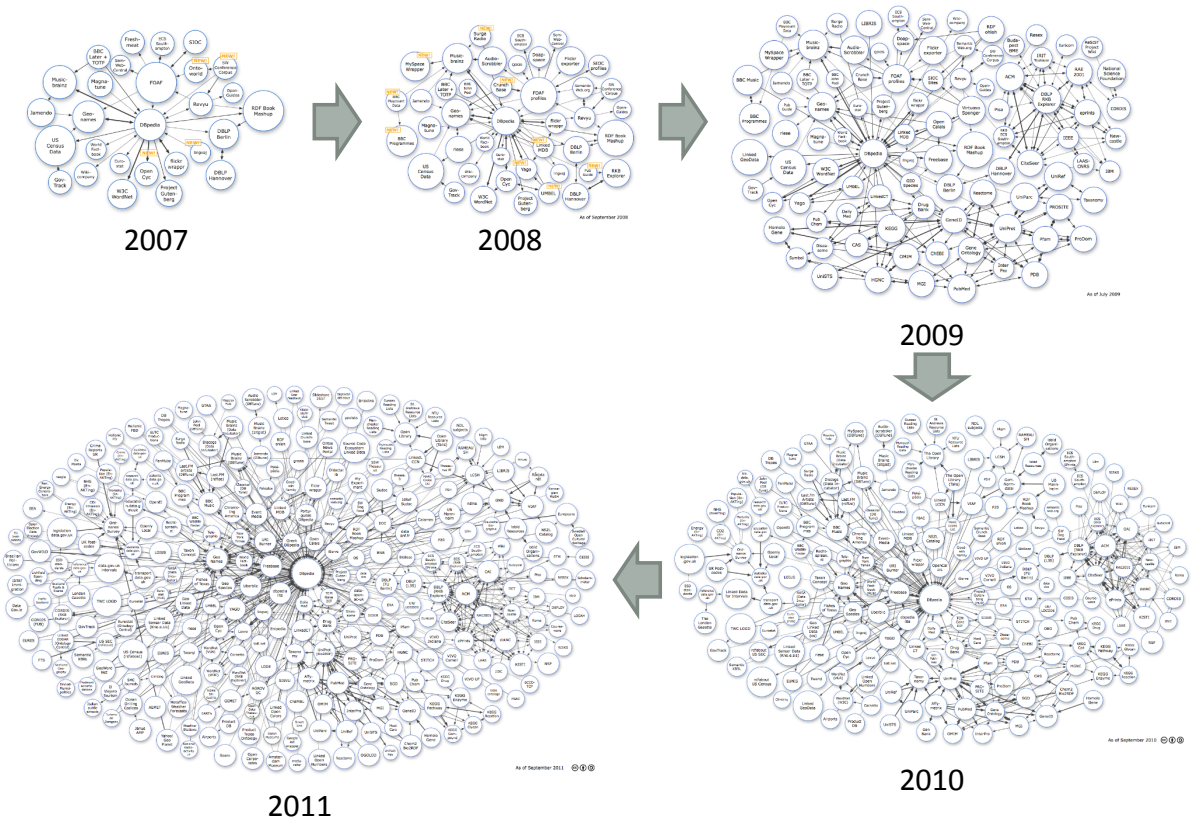


Figure 2.7: Linked Open Data cloud diagram by Cyganiak R. and Jentzsch A.

The set of interconnected public semantic databases is called "*Linked Data*"[12]. A quick look at the Linked Open Data cloud diagram by Richard Cyganiak and Anja Jentzsch[13] and its evolution from 2007 to 2011, as featured on Figure 2.7, shows that the Semantic Web is steadily growing: each node on the diagram represents a public knowledge base and each edge one or more references connecting knowledge bases to each other (*e.g.* a triple binding a local subject to a remote object).

It seems that the advantages of the Semantic Web are progressively outweighing its flaws and, as the Semantic Web grows, the associated technologies are refined and become more unavoidable.

[12]Linked Data: `http://linkeddata.org/`
[13]Linked Open Data cloud diagram: `http://lod-cloud.net/`

2.4.3.1 Resource Description Framework

The Resource Description Framework (RDF) is the W3C standard all other W3C Semantic Web standards rely on. It is a semantic data model which, like XML, is simultaneously extremely basic and extremely generic, allowing "*anyone [to] say anything about anything*": a founding principle the Semantic Web inherited from the web.

At the heart of RDF - and therefore the Semantic Web - is the notion of **triple**, also known as "*statement*". As the name suggests, a triple is made of three parts:

- a **subject**, *i.e.* a "*resource*", identified by a Uniform Resource Identifier (URI), which could represent literally anything, *e.g.* `dbpedia:Socrates` represents the Greek philosopher Socrates on DBpedia[14];
- a **predicate**, *i.e.* a "*property*" which either relates the subject to another resource, *e.g.* `rdf:type` means the subject belongs to a "*class*", or further describes the subject, *e.g.* `dbpedia:dateOfDeath` qualifies the date a person died; and
- an **object**, *i.e.* either a "*resource*" the subject is related to through the property, *e.g.* `dbpedia:Human` represents humankind, or a "*literal*" which specifies a value for the property, *e.g.* `-399` is the year Socrates died.

For instance, the triple made of `dbpedia:Socrates`, `rdf:type` and `dbpedia:Human` states that Socrates is a human being, whereas the triple made of `dbpedia:Socrates`, `dbpedia:dateOfDeath` and the literal `-399` states that Socrates died in 399BC.

There is also a special type of resource called "*blank node*" which has no URI: those nodes can only be accessed through the resources they are related to and are generally used as placeholders.

At first glance, triples seem to compose a semantic graph with resources as nodes and properties as edges. However, that vision is inaccurate in two ways: (i) properties are themselves resources and can be the subject or the object of a triple and (ii) triples can be "*reified*", *i.e.* qualified by properties, so as to add meta-information like the date or author of the triple. Still, the vision holds true enough for the official RDF Validator[15] to offer to build a graph of the validated triples.

RDF data can be formatted/serialized in a variety of ways, depending on its use. The three most well-known and popular formats are:

- RDF/XML[16], as the name suggests, is an XML syntax and, as such, is very appropriate for automated processing and data exchange between programs, but is too verbose to be practical for human writing or reading;
- Turtle is a compact textual syntax meant specifically for human production and consumption of RDF; and
- N-Triples[17] is a subset of Turtle with none of the abbreviations that make the latter compact but slow down its processing and it is quite popular as a dump format, *i.e.* when a big number of triples must be exported, backed up or imported at once.

[14]DBpedia: `http://dbpedia.org`
[15]RDF Validator: `http://www.w3.org/RDF/Validator/`
[16]RDF/XML: `http://www.w3.org/TR/REC-rdf-syntax/`
[17]N-Triples: `http://www.w3.org/TR/n-triples/`

To illustrate all three formats, consider the following example: Socrates is the master of Plato, who in turn is the master of Aristotle and all three are philosophers. Expressing this in RDF requires five triples, one to relate Socrates to Plato, another to relate Plato to Aristotle and one for each philosopher to state that he is one.

The three resources representing the three philosophers as well as the class representing all philosophers can be taken from DBpedia so as to link to that very well-known website, but, to the best of our knowledge, DBpedia does not provide a property to characterize the relation between master and pupil. We will thus create such a property, called `masterOf` in our example. Figure 2.8 is a graph representing our five triples: each triple is represented as an edge from the subject to the object. In the following listings, we have highlighted the resource and property names in bold to improve legibility and ease comparison between the three formats presented here.

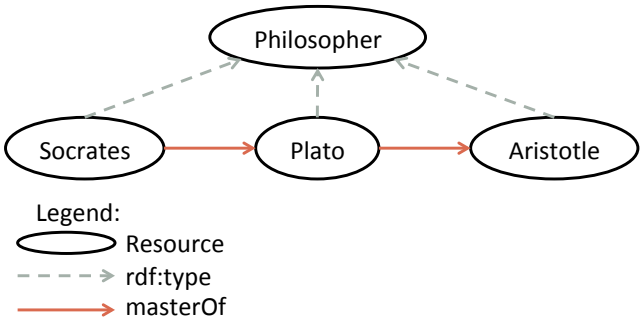


Figure 2.8: Small triple sample

The Turtle version, shown on Listing 2.1, is somewhat straightforward. It begins with the definition of a few namespaces as prefixes, so as to shorten the triples themselves. For instance, `rdf:type` will actually refer to the `type` anchor in the RDF specification. Each triple is then written as `subject␣predicate␣object`, followed by a dot which indicates its end.

Listing 2.1: Turtle example

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix dbonto: <http://dbpedia.org/ontology/> .
@prefix example: <http://example.org/rdfexample#> .

dbpedia:Socrates example:masterOf dbpedia:Plato .
dbpedia:Plato example:masterOf dbpedia:Aristotle .
dbpedia:Socrates rdf:type dbonto:Philosopher .
dbpedia:Plato rdf:type dbonto:Philosopher .
dbpedia:Aristotle rdf:type dbonto:Philosopher .
```

Triples are written the same way in N-Triples, shown on Listing 2.2, but with the full URIs, since one cannot use abbreviations like the Turtle `@prefix` statement.

Listing 2.2: N-Triples example

```
<http://dbpedia.org/resource/Socrates> <http://example.org/rdfexample#masterOf>
      <http://dbpedia.org/resource/Plato> .
<http://dbpedia.org/resource/Plato> <http://example.org/rdfexample#masterOf>
      <http://dbpedia.org/resource/Aristotle> .
<http://dbpedia.org/resource/Socrates> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      <http://dbpedia.org/ontology/Philosopher> .
<http://dbpedia.org/resource/Plato> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      <http://dbpedia.org/ontology/Philosopher> .
<http://dbpedia.org/resource/Aristotle> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
      <http://dbpedia.org/ontology/Philosopher> .
```

The RDF/XML version, shown on Listing 2.3, is significantly more verbose than the others. Prefixes can be defined like in Turtle, by using the `xmlns` attribute, which defines an XML namespace. Because an XML document is a tree and neither a graph nor a flat list of statements, the structure differs significantly: by default, properties are children nodes of their subject resource `Description` node and the object is an attribute of the property.

Listing 2.3: RDF/XML example

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:example="http://example.org/rdfexample#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://dbpedia.org/resource/Socrates">
    <example:masterOf rdf:resource="http://dbpedia.org/resource/Plato"/>
    <rdf:type rdf:resource="http://dbpedia.org/ontology/Philosopher"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Plato">
    <example:masterOf rdf:resource="http://dbpedia.org/resource/Aristotle"/>
    <rdf:type rdf:resource="http://dbpedia.org/ontology/Philosopher"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Aristotle">
    <rdf:type rdf:resource="http://dbpedia.org/ontology/Philosopher"/>
  </rdf:Description>
</rdf:RDF>
```

There are many other ways to write a series of triples in RDF/XML. For instance, Listing 2.4 is strictly equivalent to the version shown on Listing 2.3, but slightly more condensed: there is no `rdf:Description` node and resource objects are children of the property rather than attributes, which allows recursive imbrication.

Listing 2.4: RDF/XML example (alternative)

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dbonto="http://dbpedia.org/ontology/"
  xmlns:example="http://example.org/rdfexample#"
>
  <dbonto:Philosopher rdf:about="http://dbpedia.org/resource/Socrates">
    <example:masterOf>
      <dbonto:Philosopher rdf:about="http://dbpedia.org/resource/Plato">
        <example:masterOf rdf:resource="http://dbpedia.org/resource/Aristotle"/>
      </dbonto:Philosopher>
    </example:masterOf>
  </dbonto:Philosopher>
  <dbonto:Philosopher rdf:about="http://dbpedia.org/resource/Aristotle"/>
</rdf:RDF>
```

Though RDF is a semantic data model, it enforces very little in the way of semantics, presumably to remain generic enough to fit any ontology: it defines only a handful of classes and properties, the most inescapable of which is `rdf:type` and none of those classes and properties come with inference rules, which means RDF is not an ontology language. Interestingly, it does not even define the classes `Resource` and `Class`, even though their existence is induced by the `rdf:type` property.

2.4.3.2 RDF Schema

RDF Schema (RDFS) is a thin layer of semantics on top of RDF and is meant to create the most basic types of ontologies: vocabularies and taxonomies. It introduces `rdfs:Resource`, `rdfs:Literal`, `rdfs:Class` and a handful of other classes used implicitly in RDF data, as well as multiple properties, *e.g.* `rdfs:label` which defines a label to display the resource in a text. Properties defined in RDF and RDFS are not all created equal. Some carry "*inference rules*": their use as predicate implies that new triples can be inferred.

`rdfs:domain` (resp. `rdfs:range`) describes the domain (resp. range) of a property. It implies that any time a resource is the subject (resp. object) of that property, it can be inferred that said resource is of the domain (resp. range) type. For instance, `masterOf rdfs:domain Philosopher` not only states that the domain of the `masterOf` property we created is the `Philosopher` class, but that if we state that `A example:masterOf B`, then we can infer `A rdf:type Philosopher`.

`rdfs:subClassOf` is the cornerstone of any taxonomy, since it creates a hierarchy of classes. It is easily conflated with class inheritance relations found in object-oriented languages and UML models, but the mechanisms differ significantly: `rdfs:subClassOf` does not in any way restrict the classes it is applied on or the instances thereof. It simply means that any instance of the subclass can be inferred to also be an instance of the superclass. For instance, `Human rdfs:subClassOf Mortal` means precisely that from any stated `A rdf:type Human` triple can be inferred a `A rdf:type Mortal` triple.

`rdfs:subPropertyOf` is the equivalent of `rdfs:subClassOf` for properties. For instance, `motherOf rdfs:subPropertyOf ancestorOf` implies that from any stated `A motherOf B` triple can be inferred a `A ancestorOf B` triple.

To illustrate the inference mechanisms of those four properties, let us consider the following example, illustrated by a graph on Figure 2.9: we state that Socrates is the master of Plato and that Plato is the master of Aristotle, on the other hand; that our `masterOf` property has the `Philosopher` class for domain as well as range and that it is a sub-property of another one called `inspired`; and that the `Philosopher` class is a sub-class of `Human`.

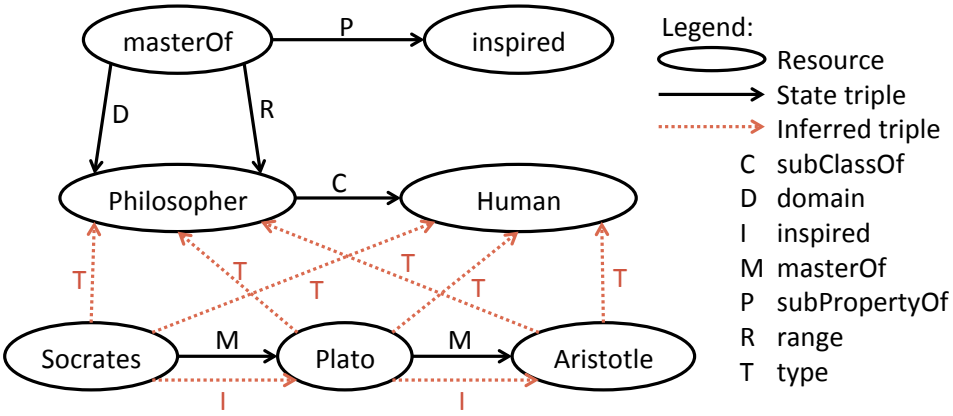


Figure 2.9: RDFS inference example

Here is what we can infer:

- since `Philosopher` is the domain of `masterOf`, then it is also the type of all its subjects, *i.e.* `Socrates` and `Plato`;
- similarly, since `Philosopher` is the range of `masterOf`, it is also the type of all its objects, *i.e.* `Plato` and `Aristotle`;
- since `Philospher` is a sub-class of `Human`, then all instances of the former are also instances of the latter, *i.e.* `Socrates`, `Plato` and `Aristotle`; and
- since `masterOf` is a sub-property of `inspired` then from the statement that Socrates is the master of Plato, we can infer that Socrates inspired Plato and the same goes for Plato and Aristotle.

More sophisticated inferences, such as transitivity - *e.g.* to infer that the ancestor of an ancestor of someone is also their ancestor - or inverse properties - *e.g.* to infer from `A motherOf B` that `B childOf A` - require more sophisticated ontology languages, such as OWL.

With only a handful of triples, it is fairly easy to draw inferences manually. For any more significant processing, or if automation is needed, there are many "*semantic reasoners*" available, notably BaseVISor [Matheus 06], CORESE [Corby 05], Fact++ [Tsarkov 06], HermiT [Motik 09], Pellet [Sirin 07] and the one included in the Jena framework [Carroll 04].

2.4.3.3 SPARQL Protocol and RDF Query Language

Once triples have been stated and/or inferred, the standard way to browse them is to use the SPARQL Protocol and RDF Query Language (SPARQL), designed and recommended by the W3C. At first glance, it seems to be for knowledge bases what SQL is for relational databases. SPARQL 1.0 crucially lacks update queries to modify data, making it very unlike SQL, but after the introduction of modification queries in SPARQL 1.1, the comparison holds. Yet, it is important to note that each query language aligns closely with the structure of the data it targets: SQL is based on relational algebra and SPARQL is built around graph patterns.

For browsing, *i.e.* setting aside update queries, SPARQL offers four basic types of queries. All four types start by matching a graph pattern with variables against the graph made of all triples. They only differ in how they handle the matches found:

- `SELECT` queries return the nodes the variables matched, for each match found;
- `CONSTRUCT` queries build a graph for each match found and return their union;
- `ASK` queries return `false` if no match is found, `true` otherwise; and
- `DESCRIBE` return information (in the form of RDF triples) about the resources identified through URIs or variable matching.

`CONSTRUCT` queries build graphs based on the results of pattern matching and, as such, they can be used as a graph rewriting tool. They are composed of two graph patterns: one simply called "*pattern*", matched against the data, and the other called "*template*", used to build new graphs based on the matches.

An analogy can be made with aspects (described in Section 2.2.2.3), comparing "*pattern*" with "*pointcuts*", to determine where the graph should be affected, and "*template*" with "*advice*", to determine how the graph should be affected. The most significant difference is that `CONSTRUCT` queries do not directly affect the graph of data they are run on, but instead build a new graph. On the one hand, it means that a `CONSTRUCT` query in and of itself cannot work as a weaving mechanism: it leaves the merging of its solution with the base graph to be done. On the other hand, it induces a lot of flexibility since the author of a query can focus on the parts that must be matched and built while ignoring the rest of the graph.

The SPARQL syntax is fairly close to that of RDF format Turtle: prefixes are set with a `PREFIX` command and triples are written in-line as `subject␣predicate␣object`. The syntax is also heavily inspired by that of SQL: keywords are written in uppercase by convention (though their case is ignored) and followed by either a single line or a block of lines delimited by brackets.

To illustrate SELECT queries, let us consider the following example: the list of monarchs who were murdered in France, like Henry IV of France. To obtain such a list requires a handful of things: a class representing monarchs, a class representing the fact of having been murdered in France and, of course, RDF data about monarchs. The DBpedia knowledge base provides all three things.

As shown on Listing 2.5, a SELECT query starts with the definition of prefixes, with the PREFIX keyword, much like in Turtle. Then the query defines which variables should be returned by the query with the SELECT keyword and a ? marking a variable as such. Finally, the WHERE keyword gives a series of triples forming the graph pattern that will be matched against the knowledge base. If we only want labels from a given language, then we need to use the FILTER keyword and utility functions to match that language, but of course such filtering is optional.

Listing 2.5: SPARQL SELECT example query

```
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX rdf:         <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:        <http://www.w3.org/2000/01/rdf-schema#>
PREFIX yago:        <http://dbpedia.org/class/yago/>

SELECT ?label
WHERE {
  ?king rdf:type dbpedia-owl:Royalty .
  ?king rdf:type yago:PeopleMurderedInFrance .
  ?king rdfs:label ?label .
  FILTER(LANGMATCHES(LANG(?label), "EN"))
}
```

label
"Henry IV of France"@en
"Charles Ferdinand, Duke of Berry"@en
"Henry III of France"@en
"John the Fearless"@en
"Louis I, Duke of Orléans"@en

Figure 2.10: DBpedia HTML result screenshot

If run on the DBpedia knowledge base, for instance using the Virtuoso SPARQL endpoint[18], this query will return Charles Ferdinand, Duke of Berry, Henry III of France, Henry IV of France, John the Fearless and Louis I, Duke of Orleans, in any order. Figure 2.10 is a screenshot of that result, returned by Virtuoso as HTML.

Now, to illustrate CONSTRUCT queries, let us consider another example, inspired by famous historical novelist Maurice Druon: he nicknamed the sons of Philip IV of France (the Fair) "*accursed kings*" and wrote a series of novels under that very name. To build a simple graph reflecting that, we need to identify Philip the Fair, find his children who became kings of France and type them AccursedKing. DBpedia provides all but the type we need, so we will create that type in our example.

As shown on Listing 2.6, CONSTRUCT queries work very much like SELECT queries, save for the main keyword, CONSTRUCT, followed by a WHERE graph pattern.

[18]DBpedia Virtuoso SPARQL endpoint: http://dbpedia.org/sparql

Listing 2.6: SPARQL CONSTRUCT example query

```
PREFIX dbpedia:     <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX example:     <http://example.org/sparqlexample#>
PREFIX rdf:         <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX yago:        <http://dbpedia.org/class/yago/>

CONSTRUCT {
  ?king rdf:type example:AccursedKing .
}
WHERE {
  ?king rdf:type yago:KingsOfFrance .
  ?king dbpedia-owl:parent dbpedia:Philip_IV_of_France .
}
```

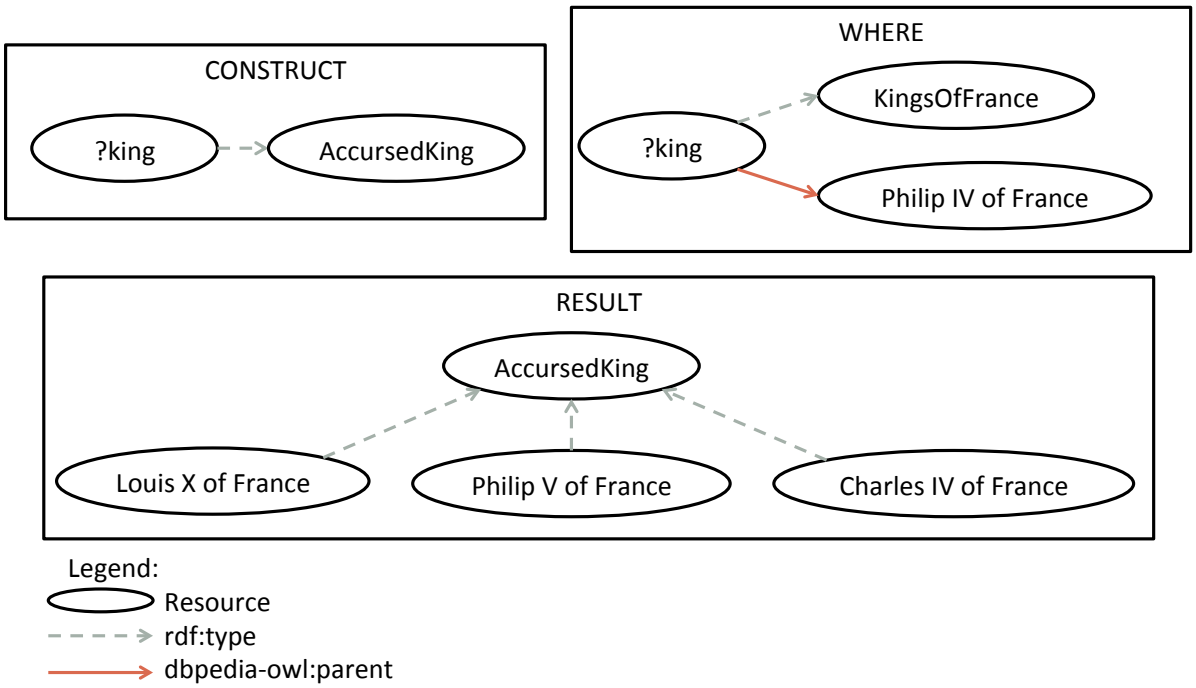


Figure 2.11: SPARQL CONSTRUCT example

If run on the DBpedia knowledge base, this query will build a graph made of three triples, binding each son of Philip the Fair - Louis X, Philip V and Charles IV - to the accursed king type we created, as illustrated on Figure 2.11.

`CONSTRUCT` queries handle variables (signaled by a `?`) and blank nodes (signaled by `_:`) differently. If a variable appears in the template (*i.e.* the `CONSTRUCT` graph pattern) and not in the pattern (*i.e.* the `WHERE` graph pattern), it will be created exactly once, no matter the number of matches found. However, if a blank node appears in the template and not in the pattern, a new blank node will be created for each match found. That fact makes blank nodes a rather easy way to generate nodes with `CONSTRUCT` queries, but there is a flaw: those nodes have no URI and thus cannot afterwards be referenced directly.

As with semantic reasoners, there are many "*SPARQL engines*" available. Some are standalone software and/or libraries, like CORESE [Corby 05], but most are provided as "*endpoints*" (*i.e.* web service that supports SPARQL) or APIs to specific "*triple stores*" (*i.e.* RDF database), as in the Jena framework [Carroll 04].

There are multiple ways to query the Linked Open Data cloud [Hartig 10], besides downloading and merging all relevant datasets locally, including:

- **Follow-up queries**: query one dataset first, adapt the next query based on the result, query the next dataset and so on. Obviously that approach requires a lot of knowledge about not only dataset structures, but also how data is spread over the datasets and a lot of work on the part of the user building the queries.
- **Federated endpoints**: some endpoints, like the one managed by OpenLink SW[19], act as a front-end to multiple datasets. That approach is a lot easier for the user, but he or she has very little control over how data is actually queried and little certainty as to its freshness.
- **Federated query processing**: "*mediators*", such as AliBaba's Federation SAIL[20], can be used to automatically dispatch a query to relevant datasets. However, it is generally no easy task to configure those mediators. [Haase 10] compares many such solutions quantitatively.

2.4.4 Discussion

We use Semantic Web technologies in three distinct ways:

- The **Conceptual Workflow Model**, defined in Chapter 3, itself is modeled in both UML (and implemented in Java in our prototype) and in RDFS, which allows reasoning on the **Conceptual Workflows** themselves.
- The **Weaving** mechanism, described in Section 4.2, is based on SPARQL `CONSTRUCT` queries, so as to leverage their flexibility as a graph rewriting method and to capitalize on the aforementioned RDFS representation of **Conceptual Workflows**.
- The **Mapping** step of the **Transformation Process** is computer-assisted thanks to domain knowledge captured in RDFS-compatible ontologies. By relying on the most popular form of ontology languages, we hope to make our process compatible with as many domains as possible.

So far, we have had no need for inference rules beyond the basic ones found in RDFS, but future extension of the system to more sophisticated ontology languages, such as OWL, is entirely conceivable.

[19] OpenLink SW SPARQL endpoint: `http://lod.openlinksw.com/sparql`
[20] AliBaba Federation SAIL: `http://www.openrdf.org/alibaba.jsp`

CHAPTER SUMMARY

The **abstraction level** of most scientific workflow models is **low**, given that domain scientists are the intended users. We identified four important features related to high abstraction level: support for **annotations**, computer assistance for workflow **composition**, **indirection** between the description and implementation of activities and **structural flexibility**. To the best of our knowledge, no existing scientific workflow model features all four.
The **Transformation Process** we are developing (i) takes inspiration from existing approaches to support **Separation of Concerns** with the notion of **Fragments**, (ii) follows the principles of **Model-Driven Engineering** and (iii) leverages **standard semantic web technologies** to process and capitalize on domain knowledge with maximum flexibility and compatibility.

CHAPTER 3
CONCEPTUAL WORKFLOW MODEL

In this Chapter, we present the scientific workflow model we defined to fulfill our first goal as defined in Section 1.6: our objective was to create a model at the **Conceptual Level** - hence the name **Conceptual Workflow** - so as to facilitate the capture of domain knowledge and know-how and increase legibility in an attempt to improve the accessibility of scientific workflows.

Conceptual Workflows are meant to be iteratively transformed into abstract workflows that can be enacted on a Distributed Computing Infrastructure (DCI). As a result, the **Conceptual Workflow Model** features:

- **Conceptual Elements**, detailed in Section 3.1 and used to describe simulations at a high level of abstraction that is computation-independent;
- **Abstract Elements** detailed in Section 3.2 and similar to the platform-independent low-level abstract workflow elements they will be converted to; and
- **Semantic Annotations** detailed in Section 3.3 and used to guide the **Transformation Process**, which is detailed in Chapter 4.

The **Conceptual Workflow Model**, described in Unified Modeling Language (UML) in Appendix B, is based on directed graphs like most scientific workflow models, including those analyzed in Section 2.1.

3.1 Conceptual Elements

3.1.1 Conceptual Workflows

Conceptual Workflows are modeled through nested directed cyclic graphs whose nodes model the components of a simulation and whose links model inter-dependencies (see Section 3.1.4). Directed Cyclic Graphs (DCGs) have been adopted because they are the base model of the majority of scientific workflow frameworks (see Section 2.1.2) and they are nested to allow the modeling of a simulation at multiple levels of abstraction as well as encapsulation. The various elements of a simulation are modeled thusly:

- input data is modeled by **Conceptual Inputs**;
- process steps are modeled by **Conceptual Functions**;
- output products are modeled by **Conceptual Outputs**; and
- dependencies between those elements are modeled by **Conceptual Links**.

For the **Transformation Process** to remain as independent as possible from existing frameworks, we also need to model artifacts from the **Abstract Level** (*e.g.* web services, files) and that is what we call **Abstract Elements**.

Because **Conceptual Workflows** contain both high-level conceptual elements and low-level abstract elements, they are graphs composed of **two sets of vertices** and **two sets of edges**, **high-level** and **low-level**. Let us define the following sets:

Set Definitions (1 of 3)

$$\begin{aligned} H_F &\text{ is the set of all Conceptual Functions} \\ H_I &\text{ is the set of all Conceptual Inputs} \\ H_O &\text{ is the set of all Conceptual Outputs} \\ H &\text{ is the set of all Conceptual Workflows} \\ L &\text{ is the set of all Abstract Elements} \\ H \cap L &= H_F \cap H_I = H_I \cap H_O = H_O \cap H_F = \varnothing \end{aligned}$$

Conceptual Workflows are nested graphs. Their definition is less straightforward than that of regular directed graphs, because nodes in a **Conceptual Function** are either simple nodes or graphs themselves. Hence the need to define **Conceptual Workflows** through recursion, with **depth** being the number of **nested levels** in a **Conceptual Workflow**:

Conceptual Workflows of depth 0

$$\begin{aligned} H_0 &\text{ is the set of all Conceptual Workflows of depth } 0 \\ H_{F0} &\text{ is the set of all Conceptual Functions of depth } 0 \\ H_0 &= H_I \cup H_O \cup H_{F0} \\ &= \{(V_H, V_L, E_H, E_L) \mid V_H = \varnothing, V_L \subset L, E_H = \varnothing, E_L \subset V_L^2\} \end{aligned}$$

Conceptual Workflows of depth $\leq n \in \mathbb{N}^*$

$$\begin{aligned} H_n &\text{ is the set of all Conceptual Workflows of depth } \leq n \\ H_{Fn} &\text{ is the set of all Conceptual Functions of depth } \leq n \\ H_n &= H_I \cup H_O \cup H_{Fn} \\ H_{Fn} &= \{(V_H, V_L, E_H, E_L) \mid V_H \subset H_{n-1}, V_L \subset L, E_H \subset V_H^2, E_L \subset V_L^2\} \end{aligned}$$

Conceptual Workflows of any depth

$$\begin{aligned} H &= \bigcup_{n \in \mathbb{N}} H_n = H_I \cup H_O \cup H_F \\ H_F &= \bigcup_{n \in \mathbb{N}} H_{Fn} \end{aligned}$$

3.1.2 Graphical Convention

The four main classes of **Conceptual Elements** are represented as illustrated on Figure 3.1:

- **Conceptual Functions** by rectangles (layered when nested);
- **Conceptual Inputs** by trapezoids with the small edge down;
- **Conceptual Outputs** by trapezoids with the small edge up; and
- **Conceptual Links** by dashed arrows.

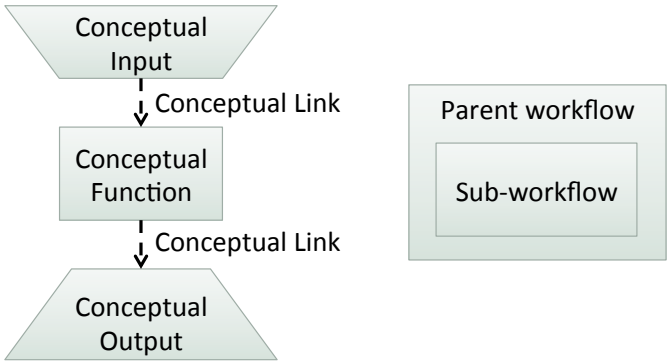
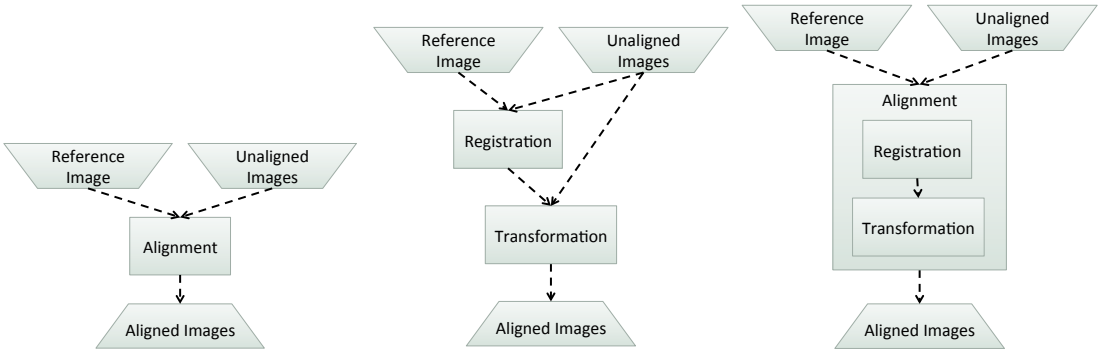


Figure 3.1: Graphical Convention - Conceptual Elements

3.1.3 Encapsulation



(a) One global step (b) Two separate steps (c) Combination

Figure 3.2: Image Spatial Alignment Process Example at Multiple Abstraction Levels

Process steps can often be divided into smaller steps or grouped into broader functions. For instance, automated image alignment can be done through registration (*i.e.* computing the transformation that most closely aligns the images) and then applying the computed transformation. As illustrated on Figure 3.2, the alignment process can be seen as one global step, two separate ones or a combination of a global step and two smaller ones. In the latter case, the smaller steps are effectively at a lower level of abstraction and should be nested into the high-level step to highlight their relation.

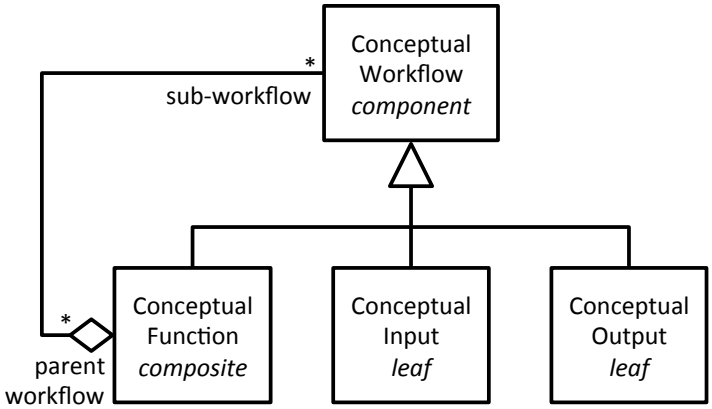


Figure 3.3: Conceptual Workflow Composite Pattern

Precisely in order to allow such modeling at multiple levels of abstraction, **Conceptual Functions** are nested: they can themselves contain **Conceptual Elements**. The meta-model (see Figure B.4) thus exhibits a "*Composite Pattern*" [Gamma 93], as described in UML on Figure 3.3, with:

- **Conceptual Workflows** as "*Components*";
- **Conceptual Functions** as "*Composites*"; and
- **Conceptual Inputs** and **Conceptual Outputs** as "*Leaves*".

When a **Conceptual Function** `P` contains a **Conceptual Workflow** `S`, `P` is relatively called the parent workflow and `S` is relatively called the sub-workflow. If `S` is also a **Conceptual Function** (as opposed to a **Conceptual Input** or **Conceptual Output**), it can in turn contain sub-workflows. The relation between parent workflows and their sub-workflows is transitive: if A is a sub-workflow of B and B is a sub-workflow of C, then A is also a sub-workflow of C.

Encapsulation (definition)

$$\begin{array}{rcl} \forall c_P & = & (V_{HP}, V_{LP}, E_{HP}, E_{LP}) \in H_{Fn}, n \in \mathbb{N}^* \\ \forall c_S & = & (V_{HS}, V_{LS}, E_{HS}, E_{LS}) \in H_{n-1} \\ c_P \prec c_S & \Leftrightarrow & c_S \in V_{HP} \end{array}$$

Encapsulation (properties)

$$\begin{array}{rrrcl} c_P \prec c_S & \Rightarrow & c_P & \text{is} & \text{called a parent workflow of } c_S \\ & \text{and} & c_S & \text{is} & \text{called a sub-workflow of } c_P \\ & \text{and} & V_{HS} & \subset & V_{HP} \setminus \{c_S\} \\ & \text{and} & V_{LS} & \subset & V_{LP} \\ & \text{and} & E_{HS} & \subset & E_{HP} \\ & \text{and} & E_{LS} & \subset & E_{LP} \end{array}$$

Where it is necessary to distinguish between direct ancestor/descendants (where the relation is defined directly) and indirect ancestors/descendants (where the relation is derived by transitivity), the direct ancestor is called the **immediate parent workflow** and the direct descendants are called the **immediate sub-workflows**.

Immediate Encapsulation

$$\begin{array}{rrcl} & \forall c_P & = & (V_{HP}, V_{LP}, E_{HP}, E_{LP}) \in H_{Fn}, n \in \mathbb{N}^* \\ & \forall c_S & \in & V_{HP} \\ \text{if} & \forall c_I & = & (V_{HI}, V_{LI}, E_{HI}, E_{LI}) \in V_{HP}, c_S \notin V_{HI} \\ \text{then} & c_P & \preceq & c_S, \\ & c_P & \text{is} & \text{called the immediate parent workflow of } c_S \\ \text{and} & c_S & \text{is} & \text{called an immediate sub-workflow of } c_P \end{array}$$

3.1.4 Conceptual Links

The components of a simulation are interconnected through either **data dependencies** meaning data produced by sources is forwarded to targets; or **control dependencies** meaning targets must wait for the sources to finish before they can start.

Section 2.1.2.3 details the differences and similarities between the two types of flow, data and control, and how they impact scientific workflow models. However, at a computation-independent level of abstraction, it is not necessarily possible to predict how a dependency between two components will be implemented and it is certainly not desirable to restrict it, since it would impede structural flexibility. Therefore, a **Conceptual Link** (*i.e.* an edge connecting two **Conceptual Workflows**), represents either (i) a set of **data dependencies**; (ii) a set of **control dependencies**; or (iii) a **combination** thereof.

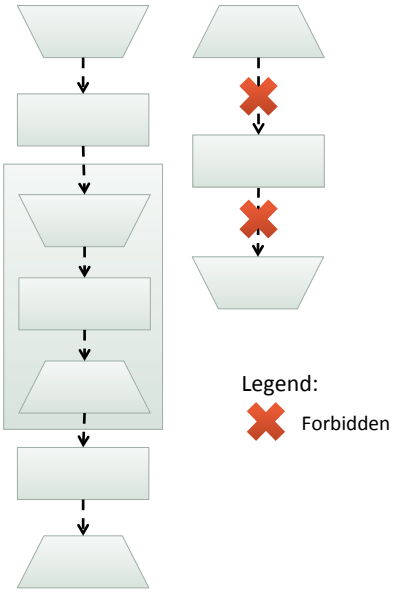


Figure 3.4: Conceptual Link Restriction

The dependencies modeled through **Conceptual Links** are transitive: to limit the links at the **Abstract Level** to pairs of **Abstract Elements** contained in the same **Conceptual Workflow** or in two **Conceptual Workflows** bound by a **Conceptual Link** would be too restrictive. Hence the notion of **conceptual path**: it is possible for **Abstract Elements** of a given **Conceptual Workflow** to depend on - *i.e.* be the target of a link bound to - **Abstract Elements** contained (i) in the same **Conceptual Workflow**; (ii) in a parent workflow or (iii) in a **Conceptual Workflow** bound by a chain of **Conceptual Links** to the one the elements are in. The symbol $\rightsquigarrow$ denotes the existence of a conceptual path between **Conceptual Workflows**:

Conceptual Path

$$
\begin{array}{lrl}
 & \forall c_P = (V_{HP}, V_{LP}, E_{HP}, E_{LP}) & \in H_{Fn}, n \in \mathbb{N}^* \\
 & \forall c_S = (V_{HS}, V_{LS}, E_{HS}, E_{LS}) & \in V_{HP} \\
 & \forall c_I & \in V_{HP} \\
 & \forall c_T & \in V_{HP} \\
(1) & c_T \in V_{HS} & \Rightarrow c_S \rightsquigarrow c_T \\
(2) & (c_S, c_T) \in E_{HP} & \Rightarrow c_S \rightsquigarrow c_T \\
(3) & c_S \rightsquigarrow c_I \text{ and } c_I \rightsquigarrow c_T & \Rightarrow c_S \rightsquigarrow c_T
\end{array}
$$

At a given nesting level, *i.e.* when source and target have the same **immediate parent workflow**, **Conceptual Links** can neither emanate from **Conceptual Outputs**, nor target **Conceptual Inputs**. However, when considering a parent workflow `P` and one of its sub-workflows `S`, then **Conceptual Links** emanating from **Conceptual Workflows** of `P` can target **Conceptual Inputs** of `S`; and **Conceptual Links** targeting **Conceptual Workflows** of `P` can emanate from **Conceptual Outputs** of `S`. Figure 3.4 illustrates both cases.

Conceptual Link Restriction

$$
\begin{aligned}
\forall c_P &= (V_{HP}, V_{LP}, E_{HP}, E_{LP}) \in H_{Fn}, n \in \mathbb{N}^* \\
\forall e &= (s,t) \in E_{HP} \\
s \in H_O &\Rightarrow \exists c_S \in V_{HP} \mid c_P \prec c_S \prec s \\
\text{and} & \\
t \in H_I &\Rightarrow \exists c_S \in V_{HP} \mid c_P \prec c_S \prec t
\end{aligned}
$$

3.2 Abstract Elements

Abstract Elements are featured in the **Conceptual Workflow Model**, despite coming from a lower level of abstraction, so that most of the **Transformation Process** takes place inside the model. This approach presents two advantages compared to a purely conceptual model:

- on the one hand, it is easier to assist the **Transformation Process** without switching to another model, especially since the process is only semi-automated;
- on the other hand, it prevents the model from being too tightly-coupled with an existing **Abstract Level** model, which would certainly complicate delegation to multiple existing frameworks.

Continuing the definitions given in Section 3.1.1, let us define sets of **Abstract Elements**:

Set Definitions (2 of 3)

$$
\begin{aligned}
L &\text{ is } \text{the set of all Abstract Elements} \\
L_A &\text{ is } \text{the set of all Activities} \\
L_P &\text{ is } \text{the set of all Ports} \\
L_{IP} &\text{ is } \text{the set of all Input Ports} \\
L_{OP} &\text{ is } \text{the set of all Output Ports} \\
L_L &\text{ is } \text{the set of all Links} \\
L_{DL} &\text{ is } \text{the set of all Data Links} \\
L_{OL} &\text{ is } \text{the set of all Order Links} \\
L_{IP} \cup L_{OP} = L_P &\text{ and } L_{IP} \cap L_{OP} = \varnothing \\
L_{DL} \cup L_{OL} = L_L &\text{ and } L_{DL} \cap L_{OL} = \varnothing \\
L_A \cup L_P \cup L_L = L &\text{ and } L_A \cap L_P = \varnothing \\
L_A \cap L_L &= \varnothing \\
L_P \cap L_L &= \varnothing
\end{aligned}
$$

3.2.1 Activities

From the viewpoint of the **Conceptual Workflow** that embeds it, an **Activity** is a black box representing an executable artifact, *e.g.* a web service, a grid job or a legacy program.

The arguments of the underlying artifact are modeled by **Input Ports** and its products by **Output Ports** associated with the **Activity**. Each **Activity** has at least one **Port**, otherwise it would be impossible to connect it to the rest of the workflow.

Some arguments may not be explicit in the underlying artifact's description. For instance, a web service might take a folder path as input and import files that are in that folder: those files are also arguments of the **Activity**, but they are implicit. The notion of **Implicit Ports** is meant to clarify those implicit relations and expose the related knowledge. By opposition, other **Ports** are **Explicit Ports**. **Implicit Ports** are a tool for the end-user to make implicit required knowledge explicit, but they are not automatically detected.

3.2.2 Specialized Activities

In addition to regular **Activities**, the **Conceptual Workflow Model** defines the following specialized ones:

- **Inputs** are **Activities** with at least one **Output Port** and no **Input Port**;
- **Outputs** are **Activities** with at least one **Input Port** and no **Output Port**; and
- **Filters** are special **Activities** implementing conditional constructs: they have one **Input Port**, two **Output Ports**: `then` and `else` and a logical condition called a **Guard**. Whenever a piece of data `d` is transferred to a **Function**, the associated **Guard** is evaluated: `d` is passed along the `then` branch if the **Guard** is `True`, along the `else` branch otherwise.

In practice, **Inputs** and **Outputs** are most often data constants or references to files, but they may also be executable artifacts (such as web services) that either only produce or only consume data.

3.2.3 Links

As explained in Section 2.1.2.3, there are two types of flow in workflows and, accordingly, there are two types of links in the Abstract part of the **Conceptual Workflow Model**:

- **Data Links** represent **data flow**, *i.e.* data transfers from a source to a target, with the target waiting for the data to execute; and
- **Order Links** represent **control flow**, *i.e.* control transfers - which can be seen as order constraints, hence the name - from a source to a target, with the target waiting until the source finishes to execute.

As they represent data transfers, **Data Links** connect **Output Ports** to **Input Ports**, whereas **Order Links** connect two **Activities** directly. Figure 3.5 describes those associations in UML.

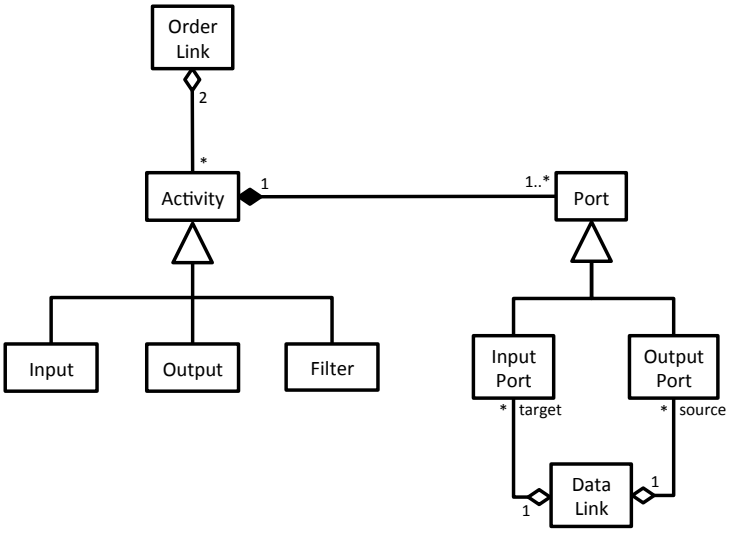


Figure 3.5: Data Links and Order Links Associations

3.2.4 Iteration Strategies

Scientific workflows are commonly used to process data sets, *i.e.* repeat the same kind of computation on collections of data items. When an **Activity** receives data of a bigger depth than it is meant to process, *e.g.* if an **Activity** treating integers (depth 0) receives a list of integers (depth 1) or a list of list of integers (depth 2) and so on and so forth, there are two main options for the enactor: (i) treat the event like a bug and inform the user that an improper type of data was provided; or (ii) loop over the **Activity** implicitly to treat all input data.

Implicit loops might make a scientific workflow slightly harder to read: how the enactor behaves depends not only on the scientific workflow at hand and the **Activities** therein, but on the input data itself. Nonetheless, it is a very useful feature for parallel data processing, to automatically dispatch data over multiple processing units in the DCI, and is thus present in multiple scientific workflow frameworks, *e.g.* Taverna [Missier 10a] and MOTEUR [Glatard 08].

If an **Activity** has multiple **Input Ports** and two or more of them receive data at higher depth than they were specified for, it is impossible to guess how data should be combined for the **Activity** to iterate over. Therefore, whenever an **Activity** has two or more **Input Ports**, it is associated with an **Iteration Strategy** that defines how data received on those multiple **Input Ports** must be combined.

An **Iteration Strategy** is a combination of all the **Input Ports** of the **Activity** via the operators:

- **Cross Product** $\otimes$ which results in every possible combination of the pieces of data received; and
- **Dot Product** $\odot$ which combines data pairs in the order in which they are received.

For instance, if an **Activity** has three **Input Ports** $\{a, b, c\}$, then $a \otimes b \otimes c$, $a \otimes (b \odot c)$, $(a \otimes b) \odot c$ and $a \odot b \odot c$ are all valid **Iteration Strategies**.

Iteration Strategies

$$\{a_i\}_{1\leq i\leq n} \otimes \{b_j\}_{1\leq j\leq m} = \{(a_i, b_j)\}_{1\leq i\leq n, 1\leq j\leq m}$$
$$\{a_i\}_{1\leq i\leq n} \odot \{b_i\}_{1\leq i\leq n} = \{(a_i, b_i)\}_{1\leq i\leq n}$$

3.2.5 Graphical Convention

The 6 types of **Abstract Elements** are represented thusly, as illustrated on Figure 3.6:

- **Activities** (including **Inputs** and **Output**) by rounded rectangles;
- **Filters** by triangles pointing downward;
- **Explicit Ports** by small rectangles;
- **Implicit Ports** by small rhombuses;
- **Data Links** by regular arrows; and
- **Order Links** by arrows with a dot head.

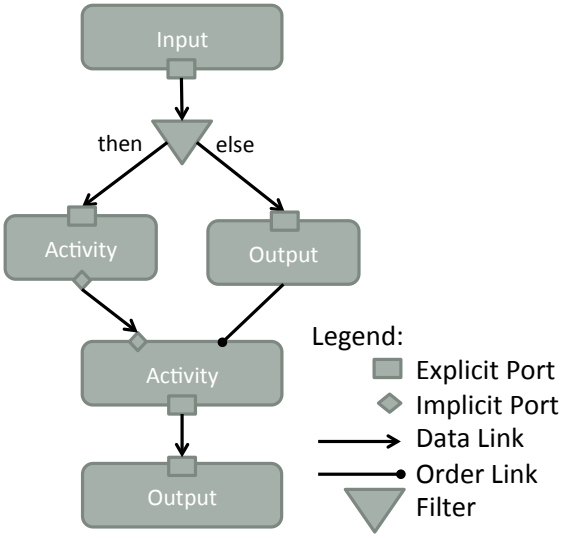


Figure 3.6: Graphical Convention - Abstract Elements

3.3 Semantic Annotations

In order to leverage Semantic Web technologies and to ensure maximum flexibility when it comes to **Semantic Annotations**, the **Conceptual Workflow Model** itself is captured in an ontology called COnceptual WORKflow (COWORK). Every other ontology used in conjunction with our system is therefore called **external ontology** by contrast with our inner model.

In order to provide assistance to the user during the **Transformation Process**, the underlying domain knowledge, as well as the technical know-how and non-functional knowledge, must be exposed. We opted to link **Conceptual Elements** and **Abstract Elements** with the domain concepts and non-functional concerns they model, as they are defined in external ontologies. As a result, many **Conceptual Elements** and **Abstract Elements**, described respectively in Section 3.1 and Section 3.2, can bear **Annotations**.

An **Annotation** is defined by three things in the **Conceptual Workflow Model**:

- its **Type** detailed in Section 3.3.1;
- its **Role**, detailed in Section 3.3.2; and
- its **Meaning**, detailed in Section 3.3.3.

3.3.1 Type

Domain ontologies used for simulation modeling, like the one defined for and used on the Virtual Imaging Platform (VIP) platform [Marion 11], often contain extensive taxonomies of the concepts the related simulations handle. In order to exploit type inference, **Annotations** are simultaneously of the type `cowork:Annotation` – defined in the COWORK ontology – and of a type defined in an external ontology.

3.3.2 Role

At a computation-independent level of abstraction, **Conceptual Elements** do not yet achieve any goals or fulfill any criteria. Therefore, at that level, **Annotations** associating domain concepts and non-functional concerns with **Conceptual Elements** are **Requirements**: they represent the objectives of the **Conceptual Elements** they annotate, rather than what the **Conceptual Elements** do.

Mapped **Conceptual Elements**, which embed sub-workflows and/or abstract workflows to fulfill their **Requirements** no longer need them. They are annotated instead with **Specifications** that describe the goals achieved and the criteria satisfied by the **Conceptual Elements** they annotate.

Abstract Elements also bear **Specifications** describing the goals they achieve and the criteria they satisfy, so that they can be suggested as suitable candidates to embed in high-level **Conceptual Workflows**, to fulfill their **Requirements**.

During the **Mapping** phase of the **Transformation Process** (detailed in Section 4.1), **Requirements** are progressively transformed into **Specifications**, as they are fulfilled.

3.3.3 Meaning

Annotations are linked to external semantic concepts through their **Type**. For the purpose of modeling a simulation, there are three categories of relevant concepts and we distinguish three different **Meanings** accordingly:

- **Functions** describe **scientific process steps**;
- **Concerns** describe **non-functional criteria**; and
- **Datasets** describe **data content and/or format**.

3.3.4 Compatibility

Annotations belong to different sets, based on their **Role** and **Meaning**:

Set Definitions (3 of 3)

$$
\begin{array}{rcl}
\Omega & \text{is} & \text{the set of all Annotations} \\
\Omega_R & \text{is} & \text{the set of all Requirements} \\
\Omega_S & \text{is} & \text{the set of all Specifications} \\
\Omega_F & \text{is} & \text{the set of all Functions} \\
\Omega_C & \text{is} & \text{the set of all Concerns} \\
\Omega_D & \text{is} & \text{the set of all Datasets} \\
\Omega_R \cup \Omega_S = \Omega & \text{and} & \Omega_F \cap \Omega_C = \Omega_C \cap \Omega_D = \Omega_D \cap \Omega_F = \varnothing
\end{array}
$$

Not every type of **Conceptual Elements** and **Abstract Elements** can be annotated with all kinds of **Annotations**.

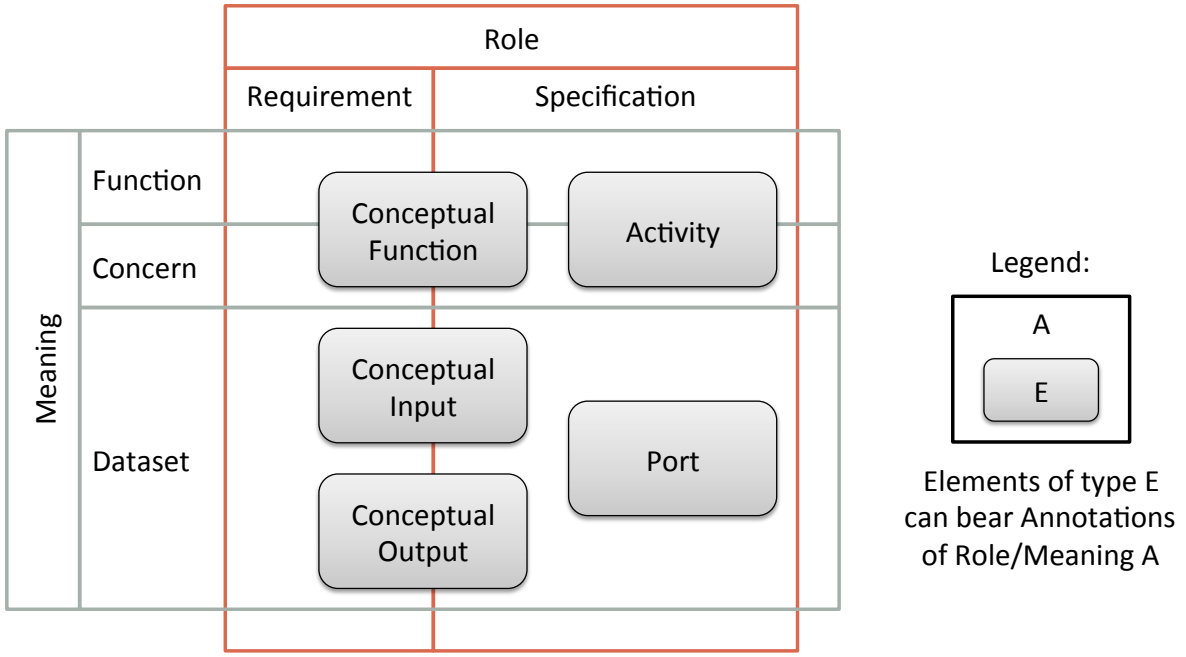


Figure 3.7: Compatibility between Elements and Annotations

As summarized in Figure 3.7:

- Elements modeling input/output data, *i.e.* **Conceptual Inputs**, **Conceptual Outputs** and **Ports**, bear only **Datasets**;
- Elements modeling processing steps, *i.e.* **Conceptual Functions** and **Activities**, bear **Functions** and **Concerns**;
- aforementioned **Conceptual Elements**, *i.e.* **Conceptual Inputs**, **Conceptual Outputs** and **Conceptual Functions**, bear either **Requirements** or **Specifications**, depending on whether they are mapped or not; and
- aforementioned **Abstract Elements**, *i.e.* **Ports** and **Activities**, bear only **Specifications**.

Let us define the function Γ which associates elements with the **Annotations** they bear and its restrictions Γ_R and Γ_S to **Requirements** and **Specifications**, respectively.

Annotation Functions

$$\begin{array}{rccc} \Gamma: & H \cup L & \longrightarrow & \Omega \\ & e & \longmapsto & a \end{array}$$

$$\begin{array}{rccc} \Gamma_R: & H & \longrightarrow & \Omega_R \\ & e & \longmapsto & r \end{array}$$

$$\begin{array}{rccc} \Gamma_S: & H \cup L & \longrightarrow & \Omega_S \\ & e & \longmapsto & s \end{array}$$

The **Annotation** compatibility restrictions can be expressed thusly:

Annotation Compatibility

$$\begin{aligned} \Gamma[H_I] = \Gamma[H_O] = \Gamma[L_P] &= \Omega_D \\ \Gamma[H_F] = \Gamma[L_A] &= \Omega_F \cup \Omega_C \\ \Gamma[H] &= \Omega_R \cup \Omega_S = \Omega \\ \Gamma[L] &= \Omega_S \end{aligned}$$

3.3.5 Graphical Convention



Figure 3.8: Graphical Convention - Annotations

There are two different representations for **Annotations**, depending on their **Role**, - in both cases, the **Type** is used as the name for the **Annotation** and the **Meaning** is not represented graphically - as illustrated by Figure 3.8: **Specifications** are represented by legend tags (*i.e.* rectangles with a line indicating where the tag is applied) and **Requirements** are represented by legend tags with a border line separating the line from the rectangle, so as to suggest some distance yet to breach.

3.4 Fragments

Conceptual Workflows are not saved as-is in the knowledge base: they are automatically saved inside **Fragments**, which may contain information about the context in which they are relevant so as to facilitate their retrieval from the knowledge base. We describe here our own definition of **Fragment** which is similar but not identical to other definitions discussed in Section 2.2.4. **Fragments** are composed of two distinct **Conceptual Workflows**: the **Blueprint** represents the content of the **Fragment** and the **Pattern** represents the context in which the **Fragment** is relevant. By default, if no **Pattern** is provided specifically, a **Conceptual Workflow** is saved as the **Blueprint** of a new **Fragment** created with an empty **Pattern**.

3.4.1 Graphical Convention

By convention, **Fragments** are represented by two dashed rectangles side by side, with the **Pattern** on the left and the **Blueprint** on the right, as shown on Figure 3.9.

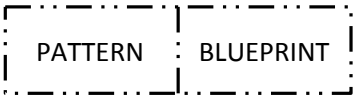


Figure 3.9: Graphical Convention - Fragments

Let us illustrate **Fragments** by considering the following example: some **Conceptual Functions** bear the annotation `CriticalStep`, which, in this case, means they are more likely to fail than others. We want to log the status of the workflow right before each critical step. One way to achieve that is to build a **Fragment** that will insert a logging step right before each **Conceptual Function** annotated with `CriticalStep`, as illustrated on Figure 3.10:

- the **Pattern** features only the **Conceptual Function** and **Annotation** we want to match as well as the **Conceptual Link** we want to bind a logging step to; and
- the **Blueprint** features the same elements, with a logging step (*i.e.* here a **Conceptual Function** annotated with a `Log` **Requirement**) inserted right before the critical step.

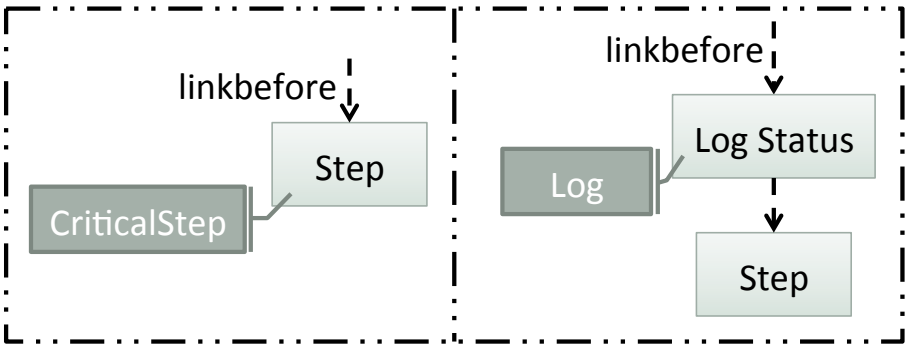


Figure 3.10: Fragment Example

3.4.2 Variables

The **Blueprint** is in every way a regular **Conceptual Workflow**, but the **Pattern** is slightly different: its elements are interpreted as variables. For instance, a **Conceptual Function** `CF` annotated with a **Function** `F` in a stand-alone **Conceptual Workflow** or in a **Blueprint** will represent a specific instance, but the same pair in a **Pattern** will be interpreted as "*any **Conceptual Function** annotated with `F`*". The names of elements in **Patterns** are thus disregarded when they are matched against a base workflow.

Another special feature of a **Conceptual Workflow** used as a **Pattern** is that it may feature links without a target or without a source, though both things cannot be missing simultaneously. The reason for that leniency is explained in Section 4.2.6.

In our example shown on Figure 3.10, the **Conceptual Function** called `Step` featured in both **Pattern** and **Blueprint** is a variable: it does not represent a specific instance of **Conceptual Function** with the rather generic name of `Step`, but instead any **Conceptual Function** annotated with `CriticalStep` and which is the target of a **Conceptual Link**. Similarly, the **Conceptual Link** called `linkbefore` represents any link targeting a **Conceptual Function** annotated with `CriticalStep`, rather than a specific instance of **Conceptual Link** called `linkbefore`. The names `Step` and `linkbefore` serve two purposes: identifying elements shared by the **Pattern** and the **Blueprint** and provide a human reader with clues as to what their use is inside the **Fragment**, though of course nothing prevents the designer of a **Fragment** from obfuscating the names of the elements therein.

CHAPTER SUMMARY

The first (out of two) **contribution** of this work is the **Conceptual Workflow Model**: a multi-level, structurally flexible and explicitly semantic scientific workflow model. It comprises **Conceptual Elements** representing domain notions, **Abstract Elements** representing implementation, **Semantic Annotations** to bind elements to domain, technical and/or non-functional ontologies and **Fragments** to support Separation of Concerns.

CHAPTER 4
TRANSFORMATION PROCESS

The transformation from a **Conceptual Workflow** to an abstract workflow involves two steps as illustrated by Figure 4.1. The first step, **Mapping**, which is overviewed in Section 4.1 and whose parts are detailed in Sections 4.2 to 4.5, is the computer-assisted transformation from a simulation, modeled at a computation-independent level of abstraction through a **Conceptual Workflow**, into an **Intermediate Representation** that contains both **Conceptual Elements** and **Abstract Elements**. The second step, **Conversion**, detailed in Section 4.6, automatically translates from the **Intermediate Representation** to a target abstract workflow language, so that execution can be delegated to an existing scientific workflow framework such as those mentioned in Section 2.1.4.

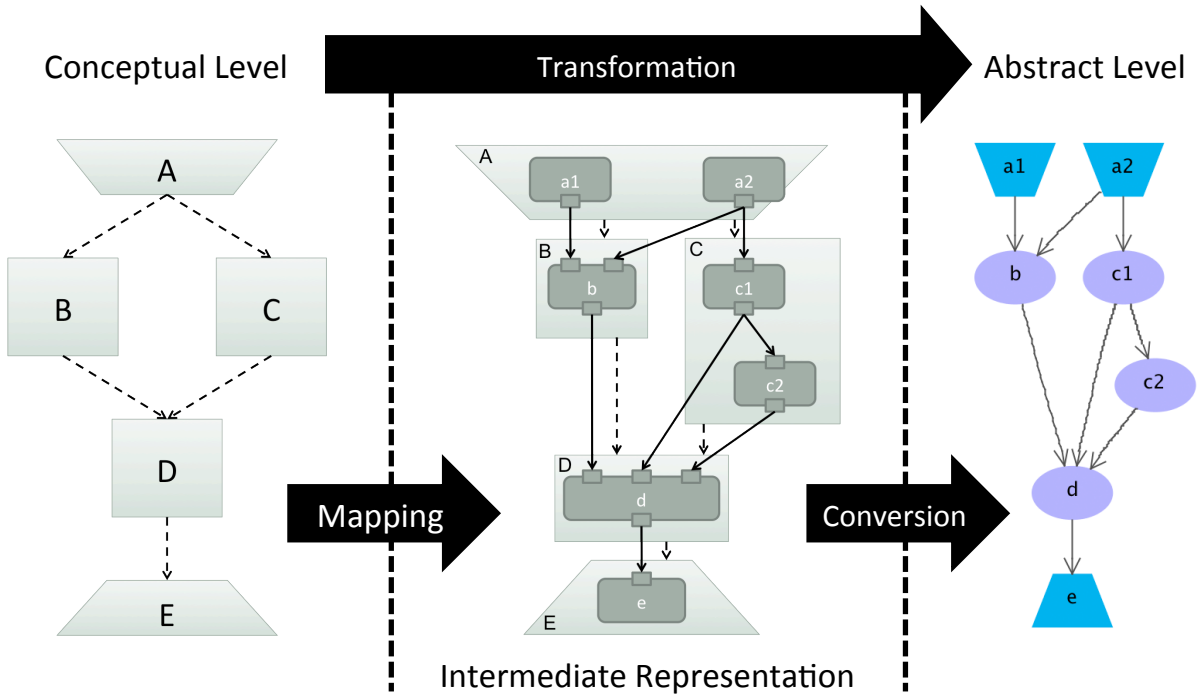


Figure 4.1: Transformation Process

4.1 Mapping

The **Mapping** step of the **Transformation Process** takes place inside the **Conceptual Workflow Model** and converts from a scientific workflow modeled at the **Conceptual Level** to an **Intermediate Representation** which lies at the **Abstract Level**. It is a semi-automated process, providing tools to assist but giving control over all decisions to the user.

4.1.1 Mechanisms

Mapping is an interactive process, driven step by step by the user, as highlighted by Figure 4.2. Its main objectives are: (i) to transform all **Requirements** into **Specifications** and (ii) to connect all **Conceptual Inputs** and **Input Ports**. The process can not be fully automated, since there is no guarantee that required **Activities** exist or that they are annotated in a way that allows their automated discovery. However, there are mechanisms in place to help users lower the abstraction level of their **Conceptual Workflow**:

- **Discovery** is the process of finding and selecting relevant **Fragments** to weave and relies on **Annotations**, as explained in Section 4.4, as well as on:
  - **Weaving** which automatically modifies the **Conceptual Workflow**, based on **Fragments** taken from the knowledge base, as explained in Section 4.2;
  - additional tools called **Erasing** and **Merging** completing **Weaving**, as explained in Section 4.3; and
- **Composition** is assisted through suggestions of links between elements of the **Conceptual Workflow** and of additional **Activities**, as explained in Section 4.5.

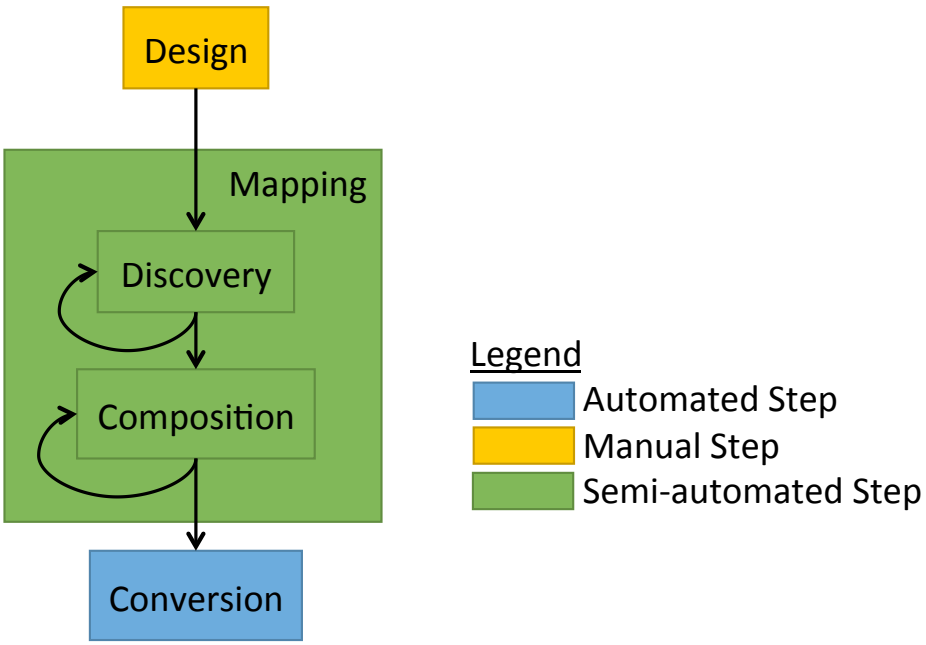


Figure 4.2: Mapping Process

4.1.2 Classification

The position of the **Mapping** model transformation in the taxonomy defined in [Czarnecki 03] and described in Section 2.3.3, is represented on Figure 4.3. It is a Feature Model instance and every node on the graph is relevant to **Mapping**:

- **Transformation Rules**: the essential transformation rule in **Mapping** is **Weaving**, described in Section 4.2, so the classification here stems from that of **Weaving**, the only difference being that some other mechanisms (*e.g.* **Merging**) are imperative;

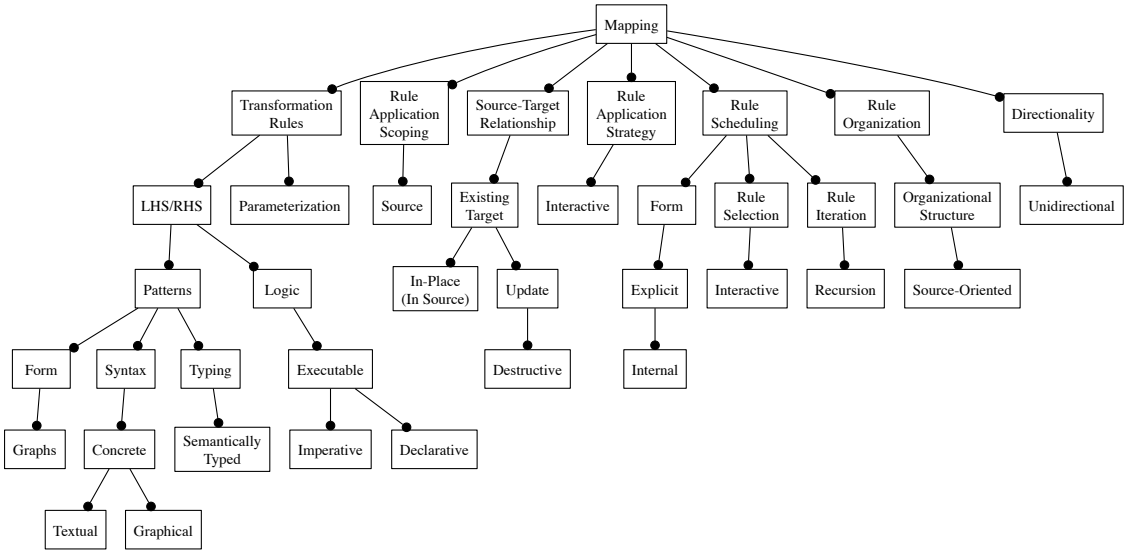


Figure 4.3: Classification of the Mapping model transformation

- **Rule Application Scoping and Strategy**: rules are interactive, *i.e.* users decide when and where to apply a rule in the **Conceptual Workflow** being mapped to an **Intermediate Representation**;
- **Source-Target Relationship**: a **Conceptual Workflow** is mapped in-place, with new elements inserted and old elements modified or even deleted, which overall amounts to destructive updating;
- **Rule Scheduling**: the rules used during **Mapping**, be it **Weaving**, **Merging**, **Erasing** or any modification the user may perform manually, are selected by the users and sometimes include internally scheduled recursive sub-rules;
- **Rule Organization**: rules are made available to the users contextually, based on the source model; and
- **Directionality**: **Mapping** is meant to lower the abstraction level of a **Conceptual Workflow** and only works in that direction.

4.2 Weaving

The **Weaving** mechanism (i) applies **Fragments**, defined in Section 3.4, on base workflows, (ii) is used as a transformation rule during **Mapping** and (iii) is itself a model transformation, hence the classification given in Section 4.2.7. It requires three things:

- a way to determine where the base process must be modified, a role played by a combination of "*join points*" and "*pointcuts*" in Aspect-Oriented Programming (AOP) and played here by the **Pattern**;
- a way to determine how the base process must be modified, a role played by the "*advice*" in AOP and played here by the combination of the **Blueprint** and **Pattern**; and
- a tool to perform the modifications, called a "*Weaver*" in AOP and described here in Section 4.2.

The modifications performed when weaving a **Fragment** depend on both **Blueprint** and **Pattern**, following those rules, summarized in Table 4.1:

- elements that are featured in the **Blueprint** but not in the **Pattern** will be **generated**: they will be created inside the base workflow;
- elements that are featured in the **Pattern** but not in the **Blueprint** will be **deleted**: they will be removed from the base workflow;
- elements that are featured in **both** will be **preserved**: they might be slightly modified in the base workflow (*e.g.* changing the target of a **Conceptual Link**), but they will not be destroyed; and
- elements that are only featured in the base workflow and in neither the **Blueprint** nor the **Pattern** will be left **untouched**: they will not be modifed in any way.

Table 4.1: Pattern/Blueprint combinations

Pattern	Blueprint	Resulting Graph
✓	✓	Preserved
	✓	Generated
✓		Deleted
		Untouched

The comparison of elements between **Blueprint** and **Pattern** is based on names: if there is a **Conceptual Input** called `CI` in the **Blueprint** and another one called the same in the **Pattern**, they will be interpreted as being the same element (in the aforementioned rules), even though they are necessarily different instances of **Conceptual Input**. Specifically for that reason, all links may bear names in **Fragments**.

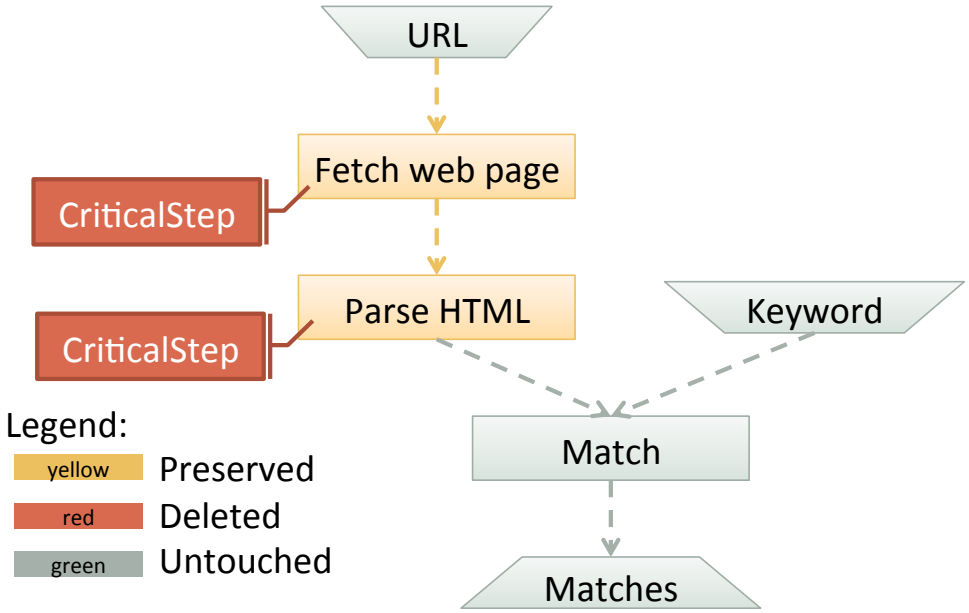


Figure 4.4: Weaving Example - Base workflow

Let us illustrate **Weaving** with the example **Fragment** defined in Section 3.4 and shown on Figure 4.5 as well as a simple base workflow, shown on Figure 4.4, which describes the process of fetching a web page and matching its contents against an input keyword. The **Annotation** `CriticalStep` is used on two steps in the workflow which are particularly likely to fail, `Fetch web page` and `Parse HTML`.

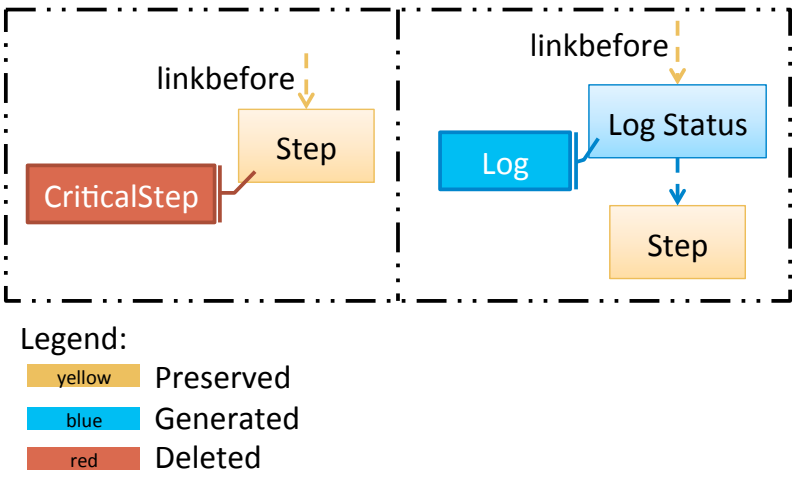


Figure 4.5: Weaving Example - Fragment

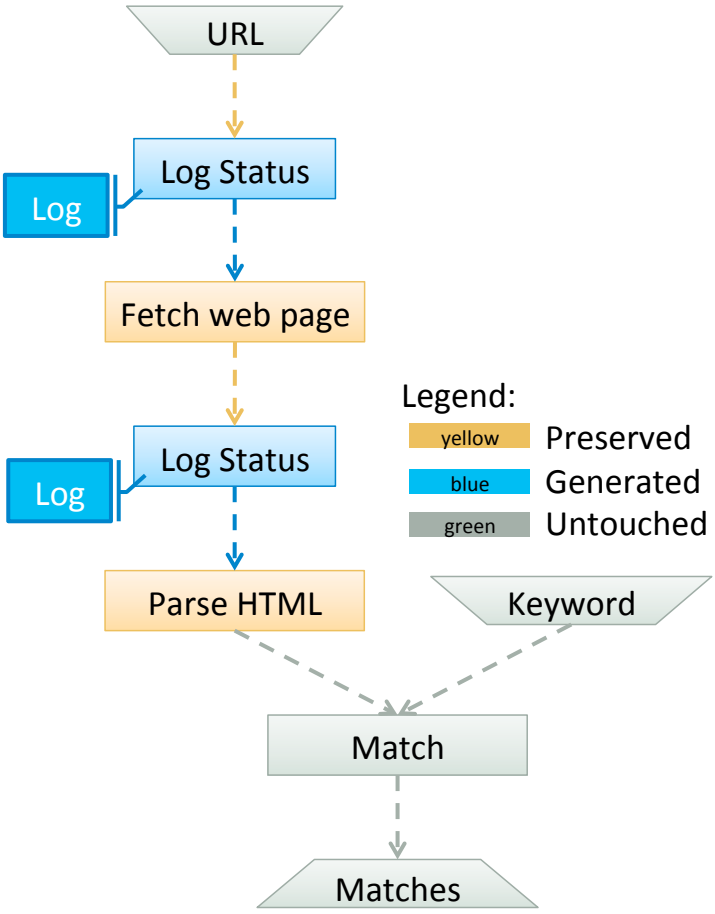


Figure 4.6: Weaving Example - Result

Figure 4.6 shows the result of **Weaving** the example **Fragment** of Figure 4.5 into the base workflow of Figure 4.4. All three figures are color coded thusly:

- **preserved elements**, *i.e.* the `Step` **Conceptual Functions**, their matches `Fetch web page` and `Parse HTML` and their incoming **Conceptual Links** are in yellow;
- **generated elements**, *i.e.* the `Log Status` **Conceptual Functions** as well as their `Log` **Annotations** and outgoing **Conceptual Links**, are in blue;
- **deleted elements**, *i.e.* the `CriticalStep` **Annotations** are in red; and
- **untouched elements** are in green.

4.2.1 Steps

As is clear from the modeling of the process in our own **Conceptual Workflow Model**, shown on Figure 4.7, **Weaving** comprises 5 steps:

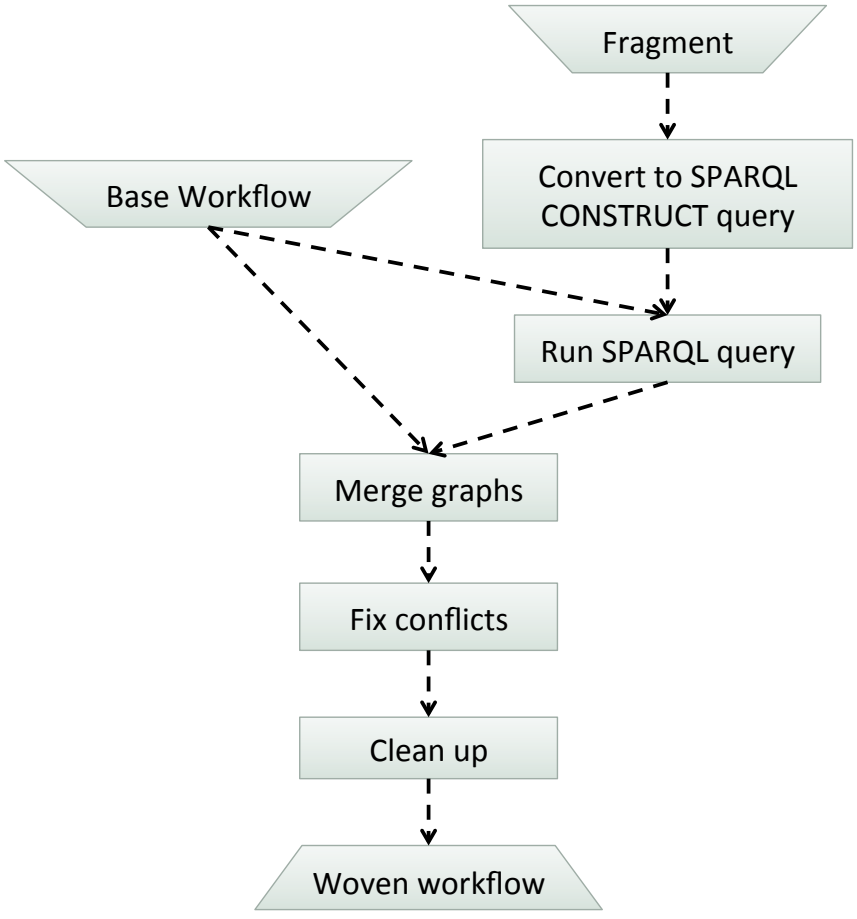


Figure 4.7: Weaving Process

1. the **Fragment** is converted to a SPARQL Protocol and RDF Query Language (SPARQL) `CONSTRUCT` query, as explained in Section 4.2.2;
2. the resulting query is run against the base **Conceptual Workflow**, as explained in Section 4.2.3;

3. the graph result of the query is merged with that of the base **Conceptual Workflow**, as explained in Section 4.2.3;

4. conflicts introduced at the previous step are fixed, as explained in Section 4.2.4; and then

5. elements present in the **Pattern** but not in the **Blueprint** are deleted and Uniform Resource Identifiers (URIs) are generated for blank nodes, as explained in Section 4.2.5.

In many cases, after **Weaving**, depending on how new elements are bound to existing ones, as detailed in Section 4.2.6, users might want to employ the **Merging** and/or **Erasing** tools, respectively described in Section 4.3.1 and Section 4.3.2, to reach their goals.

4.2.2 Fragment to SPARQL Conversion

SPARQL is the standard query language for Resource Description Framework (RDF) knowledge bases. Since it is built around the notion of graph pattern matching, it is a very good match for **Fragments**. In particular, `CONSTRUCT` queries (*cf.* Section 2.4.3.3) can be leveraged to implement **Weaving** fairly easily.

The conversion from a **Fragment** to a SPARQL `CONSTRUCT` query is straightforward. The **Pattern** is converted into the `CONSTRUCT` part of the query and the **Blueprint** into the `WHERE` part. Triples are converted according to the following rules:

- properties are kept as-is;
- resources belonging to the COnceptual WORKflow (COWORK) ontology or to an external one are kept as-is;
- other resources (*i.e.* instances pertaining to the **Fragment**) are transformed into SPARQL variables (*i.e.* names preceded by `?` markers) by default; and
- every resource in the **Blueprint** whose name does not exist in the **Pattern** is transformed into a blank node (*i.e.* a placeholder name preceded by `_:`). This use of blank nodes ensures that elements present in the **Blueprint** but not in the **Pattern** will be generated once for each match.

However, not all triples are converted, as some would hinder pattern matching or produce garbage triples in the resulting workflow: In particular, `rdfs:label` declarations - which specify the names of resources and are used to compare the elements between **Pattern** and **Blueprint** - are converted neither for **Pattern** resources (otherwise they would prevent matching when names differ even slightly) nor for resources the **Blueprint** shares with the **Pattern** (otherwise preserved elements would be systematically renamed).

To illustrate the conversion from **Fragments** to SPARQL `CONSTRUCT` queries, let us reconsider the example **Fragment** first introduced in Section 3.4.1 and shown on Figure 4.5: **Weaving** this **Fragment** amounts to inserting a **Conceptual Function** annotated with a `Log` **Requirement** before every **Conceptual Function** which bears a `CriticalStep` **Requirement**.

This example uses the following prefixes:

- `rdf` is the standard specification of RDF;
- `rdfs` is the standard specification of RDF Schema (RDFS);
- `cowork` is our COWORK ontology;

- `ex` is the namespace of the **Fragment** example; and
- `domain` serves as an example of external domain ontology.

In both Listing 4.1 (the example **Fragment**) and Listing 4.2 (the result of converting said **Fragment** to SPARQL), names are abbreviated for the listings to fit side by side (for instance `Blueprint` is abbreviated as `Bluep`) and lines are aligned so the line on the right-hand listing is the conversion of the one on the left, save for lines 11, 29 and 41 which exist only in SPARQL and have no equivalent in the original **Fragment**. The unabbreviated listings are given in Appendix C.

Listing 4.1: Fragment to SPARQL Conversion - Fragment Example (abbreviated)

```
1  @prefix rdf: <.../22-rdf-syntax-ns#> .
   @prefix rdfs: <.../rdf-schema#> .
3  @prefix cowork: <.../cowork.rdfs#> .
   @prefix ex: <.../fragmentexample#> .
5  @prefix remote: <.../remoteontologyexample#> .

7  ex:Fragment1 rdf:type cowork:Fragment .
   ex:Fragment1 cowork:hasBluep ex:Bluep1 .
9  ex:Fragment1 cowork:hasPattern ex:Pattern1 .
   ex:Fragment1 rdfs:label "Example" .
11 ## no equivalent
   ex:Bluep1 rdf:type cowork:ConcepFunc .
13 ex:Bluep1 cowork:contains ex:ConcepFunc2 .
   ex:Bluep1 cowork:contains ex:ConcepFunc3 .
15 ex:Bluep1 rdfs:label "Example" .
   ex:ConcepFunc2 rdf:type cowork:ConcepFunc .
17 ex:ConcepFunc2 rdfs:label "Log␣Status" .
   ex:ConcepFunc2 cowork:hasReq ex:Function1 .
19 ex:Function1 rdf:type cowork:Function .
   ex:Function1 rdf:type remote:Log .
21 ex:ConcepFunc3 rdf:type cowork:ConcepFunc .
   ex:ConcepFunc3 rdfs:label "Step" .
23 ex:ConcepLk2 cowork:hasTarget ex:ConcepFunc2 .
   ex:ConcepLk2 rdf:type cowork:ConcepLk .
25 ex:ConcepLk2 rdfs:label "linkbefore" .
   ex:ConcepLk3 cowork:hasSource ex:ConcepFunc2 .
27 ex:ConcepLk3 cowork:hasTarget ex:ConcepFunc3 .
   ex:ConcepLk3 rdf:type cowork:ConcepLk .
29 ## no equivalent
   ex:Pattern1 rdf:type cowork:ConcepFunc .
31 ex:Pattern1 cowork:contains ex:ConcepFunc1 .
   ex:Pattern1 rdfs:label "Example" .
33 ex:ConcepFunc1 rdf:type cowork:ConcepFunc .
   ex:ConcepFunc1 rdfs:label "Step" .
35 ex:ConcepFunc1 cowork:hasReq ex:Concern1 .
   ex:Concern1 rdf:type cowork:Concern .
37 ex:Concern1 rdf:type remote:CriticalStep .
   ex:ConcepLk1 cowork:hasTarget ex:ConcepFunc1 .
39 ex:ConcepLk1 rdf:type cowork:ConcepLk .
   ex:ConcepLk1 rdfs:label "linkbefore" .
41 ## no equivalent
```

Listing 4.2: Fragment to SPARQL Conversion - Result Query (abbreviated)

```
PREFIX rdf: <.../22-rdf-syntax-ns#>                                  1
PREFIX rdfs: <.../rdf-schema#>
PREFIX cowork: <.../cowork.rdfs#>                                    3
PREFIX ex: <.../fragmentexample#>
PREFIX remote: <.../remoteontologyexample#>                          5

## ignored                                                           7
## ignored
## ignored                                                           9
## ignored
CONSTRUCT {                                                          11
  ?Example rdf:type cowork:ConcepFunc .
  ?Example cowork:contains _:LogStatus .                             13
  ?Example cowork:contains ?Step .
## ignored                                                           15
  _:LogStatus rdf:type cowork:ConcepFunc .
  _:LogStatus rdfs:label "Log␣Status" .                              17
  _:LogStatus cowork:hasReq _:Func1 .
  _:Func1 rdf:type cowork:Func .                                     19
  _:Func1 rdf:type remote:Log .
  ?Step rdf:type cowork:ConcepFunc .                                 21
## ignored
  ?linkbefore cowork:hasTarget _:LogStatus .                         23
  ?linkbefore rdf:type cowork:ConcepLk .
  ?linkbefore rdfs:label "linkbefore" .                              25
  _:ConcepLk3 cowork:hasSource _:LogStatus .
  _:ConcepLk3 cowork:hasTarget ?Step .                               27
  _:ConcepLk3 rdf:type cowork:ConcepLk .
} WHERE {                                                            29
  ?Example rdf:type cowork:ConcepFunc .
  ?Example cowork:contains ?Step .                                   31
## ignored
  ?Step rdf:type cowork:ConcepFunc .                                 33
## ignored
  ?Step cowork:hasReq ?Concern1 .                                    35
  ?Concern1 rdf:type cowork:Concern .
  ?Concern1 rdf:type remote:CriticalStep .                           37
  ?linkbefore cowork:hasTarget ?Step .
  ?linkbefore rdf:type cowork:ConcepLk .                             39
## ignored
}                                                                    41
```

The triples binding **Pattern** and **Blueprint** to the **Fragment**, *i.e.* lines 7-10 of Listing 4.1, are ignored during conversion. Indeed, the **Fragment** is woven into a **Conceptual Workflow**, not another **Fragment**. Triples specifying names, through the `rdfs:label` property, of resources present in the **Pattern**, *i.e.* lines 15, 22, 32, 34 and 40 of Listing 4.1, are not converted.

The **Blueprint** and **Pattern** automatically bear the same name (that of the **Fragment**). Indeed, they both represent the base workflow in which the **Fragment** is woven. All resources present in both **Blueprint** and **Pattern**, including themselves, or only in the **Pattern**

are converted into variables bearing either the value of their `rdfs:label` property, if it exists, (*e.g.* `ex:Pattern1 rdfs:label "Example"` implies that `Example` is the name of `ex:Pattern1`) or the local name (*e.g.* the local name of `ex:Concern1` is `Concern1`):

- `ex:Pattern1` and `ex:Blueprint1` $\Longrightarrow$ `?Example`
- `ex:ConceptualFunction1` and `ex:ConceptualFunction3` $\Longrightarrow$ `?Step`
- `ex:ConceptualLink1` and `ex:ConceptualLink2` $\Longrightarrow$ `?linkbefore`
- `ex:Concern1` $\Longrightarrow$ `?Concern1`

Resources present only in the **Blueprint** are transformed into blank nodes, based on their name (again, label or local name):

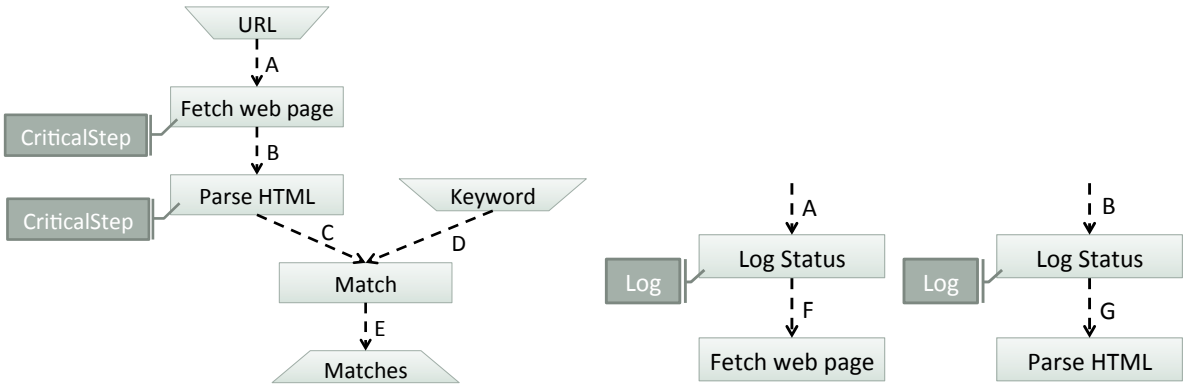
- `ex:ConceptualFunction2` $\Longrightarrow$ `_:LogStatus`
- `ex:Function1` $\Longrightarrow$ `_:Function1`
- `ex:ConceptualLink3` $\Longrightarrow$ `_:ConceptualLink3`

Apart from the conversions into variables/blank nodes, the aforementioned skipped lines and the positioning of **Pattern** (resp. **Blueprint**) triples inside the `CONSTRUCT` (resp. `WHERE`) brackets, the rest is left as it is in Turtle. For instance, `ex:Concern1` on line 37 of Listing 4.1 becomes `?Concern1` on line 37 of Listing 4.2.

4.2.3 SPARQL query

Once the query has been automatically generated, it is run against the base workflow the **Fragment** is woven onto: the SPARQL query engine is given the SPARQL `CONSTRUCT` query resulting from the **Fragment** conversion and the triple graph of the base **Conceptual Workflow** as a base model in which to execute the query.

The query produces one result graph which, if there were two or more matches, is the union of the graphs constructed (one for each match found). That result graph contains only the triples generated by the query and none of those that are in the base workflow. To obtain a workable result, at the cost of generating conflicts as explained in the following Section 4.2.4, the result graph is merged with the base workflow by a direct graph union (*i.e.* the union graph contains all the triples of both graphs).



(a) Base workflow (b) Query result

Figure 4.8: Fragment SPARQL Query - Example

For instance, in our running example, executing the query obtained in the previous section against the example base workflow shown on Figure 4.8a would produce a result graph with those two **Conceptual Functions** preceded by a `Log Status` **Conceptual Function** annotated with `Log`, as shown on Figure 4.8b. Letters on both figures are meant to identify **Conceptual Links**, especially those shared by both graphs. The union of those two graphs, the base workflow and the query result, cannot be represented as a **Conceptual Workflow**, because **Conceptual Links** `A` and `B` both would end up with multiple targets, which violates the **Conceptual Workflow Model**. This is a typical example of conflict.

4.2.4 Conflicts

Conflicts in the graphs resulting from the union of base workflows and query results stems from preserved links: links present in both **Pattern** and **Blueprint** which will, more often than not, be switched to new sources and/or targets by the **Weaving** process. The problem is that running the SPARQL `CONSTRUCT` and mixing the result graph with the base workflow will not remove anything from the original workflow: the modified links will thus have multiple sources and/or targets, which conflicts with the **Conceptual Workflow Model**.

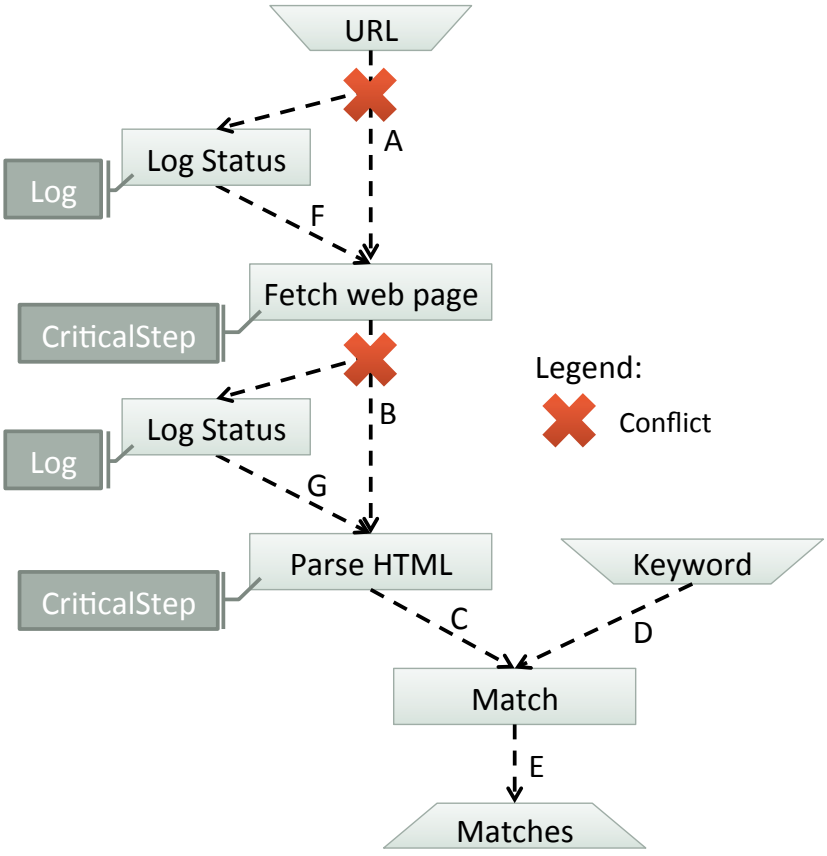


Figure 4.9: Fragment SPARQL Query - Conflicts

In our running example, the links called `A` and `B` in Figure 4.8 are preserved because they match with `linkbefore` in the **Fragment** which is shared by both **Blueprint** and **Pattern**. They are also modified in that they are given a new target. Both thus generate conflicts as highlighted by Figure 4.9.

Detecting conflicts is as simple as listing all resources that have two or more distinct sources (resp. targets). Fixing them is a little more involved; there are three cases:

- if the target (resp. source) of the link was switched to a preserved element in the **Fragment**, the link has two named targets (as opposed to blank nodes) in the union result graph, fixing then requires comparing the new graph with the original and getting rid of the target (resp. source) statement that exists in both;
- if the target (resp. source) of the link was switched to a generated element, then only the target (resp. source) statements with a blank node as object are relevant and the one statement with a named object can be safely deleted; and
- if the link ends up with two or more blank node targets (resp. source) then it must be split into as many links as there are blank node targets (resp. sources).

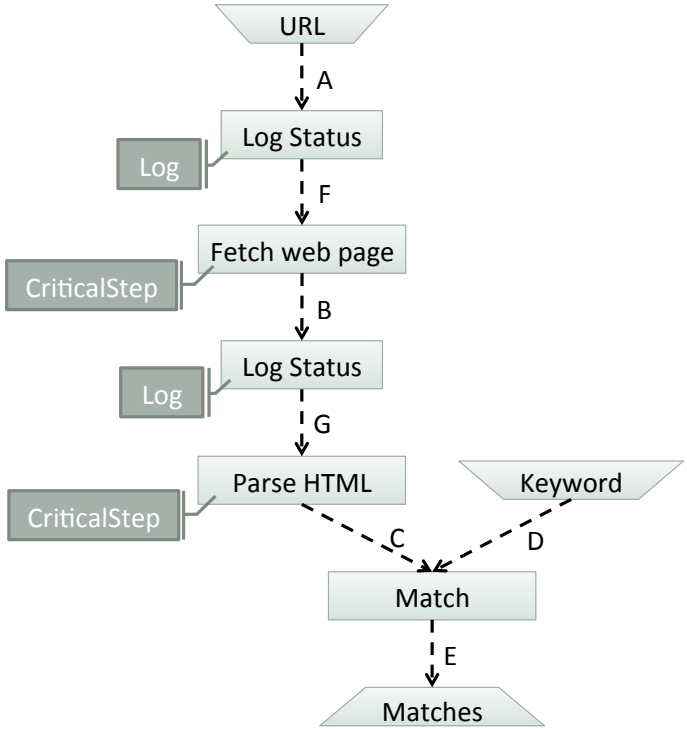


Figure 4.10: Fragment SPARQL Query - Conflicts Fixed

In our running example, the links `A` and `B` have two targets each: one is a generated `Log Status` blank node and the other is a named **Conceptual Function** (respectively `Fetch web page` and `Parse HTML`). The statements binding them to named nodes can safely be removed automatically and that results in the **Conceptual Workflow** shown on Figure 4.10. All that is left is a little cleanup, notably to get rid of deleted elements and to transform blank nodes into named ones.

4.2.5 Clean-up

The first step to clean up the workflow is to generate URIs for the blank nodes generated during **Weaving**, if any. Indeed, blank nodes work fine as a tool to generate new elements with SPARQL `CONSTRUCT` queries, but having no URI makes them hard to manipulate: one has to go through another resource they are bound to by some property. However, naming a node after creating it as a blank node is not a feature commonly available in Semantic Web libraries.

Instead, a new named node is created for each blank node, replicating all its statements, and then the blank nodes are deleted.

The second step is to get rid of deleted elements, *i.e.* elements that were present in the **Pattern** but not in the **Blueprint**. A simple way to do this is to identify those elements during the SPARQL query conversion - when **Pattern** and **Blueprint** elements are compared to determine which elements are converted to variables and which to blank nodes - and annotate them with a dedicated statement, such as `rdfs:comment "TO_REMOVE"`. Then, after querying, mixing the result graph with the base one and cleaning up URIs, it is very easy to remove all the subjects of those statements.

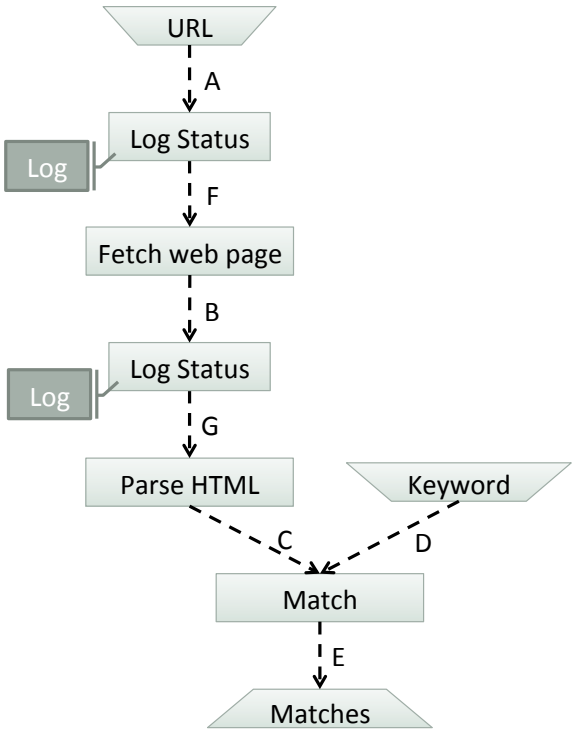


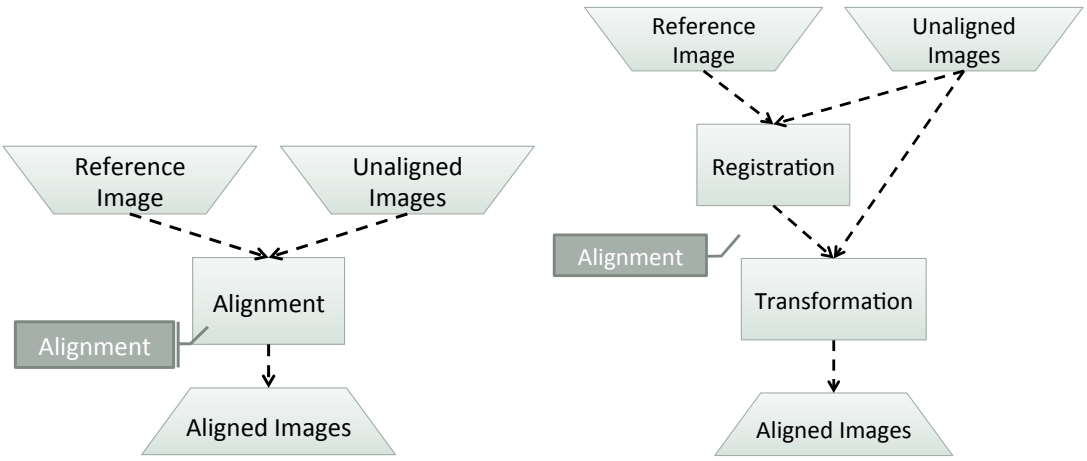
Figure 4.11: Fragment SPARQL Query - Final Result

On our running example, the only deleted element is the `CriticalStep` **Requirement**, but it is matched twice; both are thus removed and the final result is shown on Figure 4.11.

4.2.6 Binding

Elements present in the **Blueprint** but not in the **Pattern** are generated in the workflow resulting from the **Fragment Weaving** process. If they are only bound to the **Blueprint** itself or only to elements that are themselves generated as well, then they will be created disconnected from all other elements in the base workflow.

Let us consider the following example: the process of aligning images is most often realized in two steps, by first computing the best transformation from each image to a reference image - a process generally called "*registration*" - and then by applying that transformation to each image. Figure 4.12 shows a typical use of **Weaving**: the **Conceptual Workflow** on the left has only one **Conceptual Function** bearing an **Annotation** `ex:Alignment` which means it performs an image spatial alignment and the one on the right is the ideal result of **Weaving** with the alignment step replaced by the aforementioned two and the **Requirement** transformed into a **Specification** for the entire **Conceptual Workflow**.



(a) Base workflow (b) Desired result

Figure 4.12: Binding Example - Alignment Process

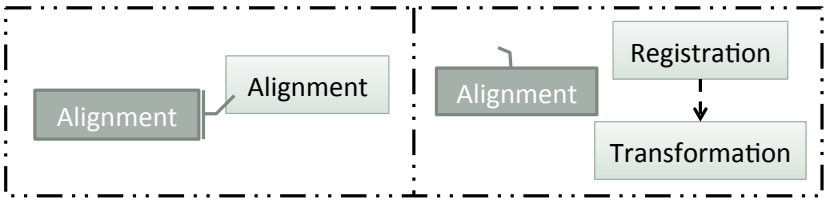


Figure 4.13: Binding Example - Unbound Fragment

It would be tempting to try to achieve that result with the simplest **Fragment**, shown on Figure 4.13: on the one hand, the **Pattern** contains only the **Conceptual Function** we want to replace; on the other hand, the **Blueprint** contains only the small sub-workflow we want to replace the aforementioned **Conceptual Function** with.

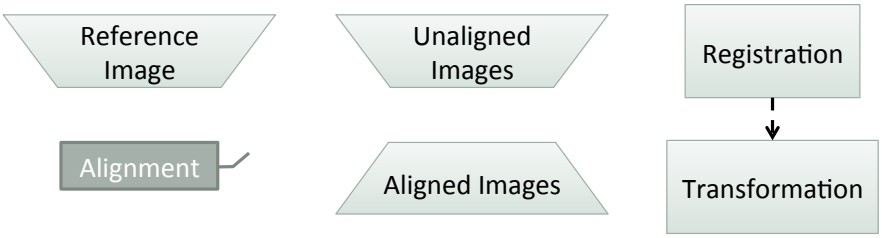


Figure 4.14: Binding Example - Unbound Weaving Result

Unfortunately, this would not achieve the desired result: the generated elements would not be bound to anything in the result workflow and would just float around, as shown on Figure 4.14. To avoid this loss of information, generated elements (*i.e.* present in the **Blueprint** but not in the **Pattern**) must be bound to preserved elements (*i.e.* present in both **Pattern** and **Blueprint**). How **Weaving** behaves depends on the type of preserved element used to bind the generated elements to, whether they are nodes or links.

4.2.6.1 Node-bound Weaving

Node-bound **Weaving** happens when generated elements are contained in a preserved node. One way to use node-bound **Weaving** in our running example is to put the `Registration` and `Transformation` **Conceptual Functions** inside the `Alignment` one, instead of deleting the latter, as shown on Figure 4.15.

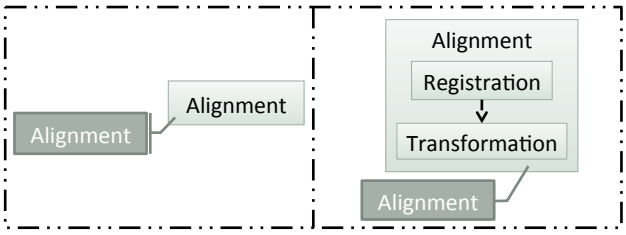


Figure 4.15: Binding Example - Node-bound Fragment

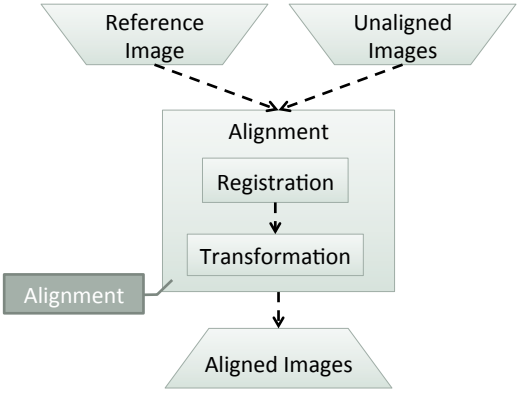


Figure 4.16: Binding Example - Node-bound Weaving Result

The new elements are then generated inside the node(s) matched to the ones that contained them in the **Fragment**, as illustrated by Figure 4.16. Though meaningful, this result is not the desired one. The differences can either be fixed manually or by using the **Erasing** mechanism described in Section 4.3.2.

4.2.6.2 Link-bound Weaving

When node-bound **Weaving** is unsuitable, *e.g.* when there is no suitable node to use for that purpose, link-bound **Weaving** can be used instead by binding the generated elements to preserved links which will necessarily be modified, *i.e.* either their source or target will be switched from whatever it was to the generated element. One way to use link-bound **Weaving** in our running example is to bind the registration and transformation **Conceptual Functions** to the same links binding the alignment one, as shown on Figure 4.17.

If the number of links in the **Pattern** is exactly the same as in the base workflow, link-bound **Weaving** works as expected, but if, as in our example, there are multiple matches, then different elements will be generated for each match, as shown on Figure 4.18. Though it might make sense in some instances to generate different elements for each path in the base workflow, *e.g.* to log data transfers, it is not the desired result in our example. The differences can either be fixed manually or by using the **Merging** mechanism described in Section 4.3.1.

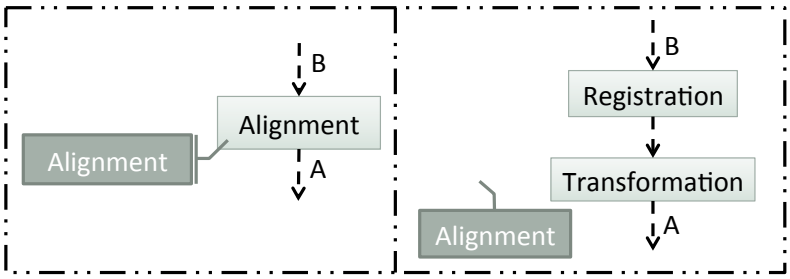


Figure 4.17: Binding Example - Link-bound Fragment

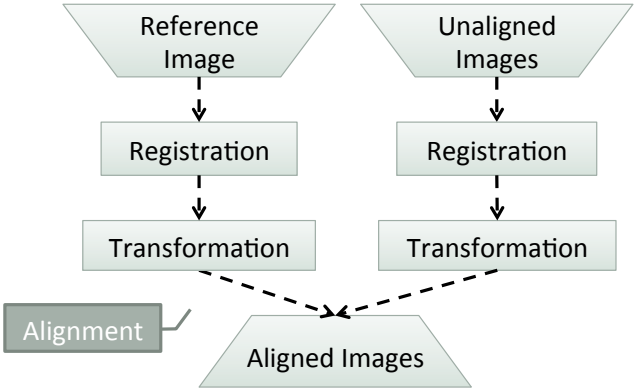


Figure 4.18: Binding Example - Link-bound Weaving Result

4.2.7 Classification

The position of the **Weaving** model transformation in the taxonomy defined in [Czarnecki 03] and described in Section 2.3.3, is represented on Figure 4.19. Like the classification of **Mapping**, it is a Feature Model instance and every node on the graph is relevant to **Weaving**:

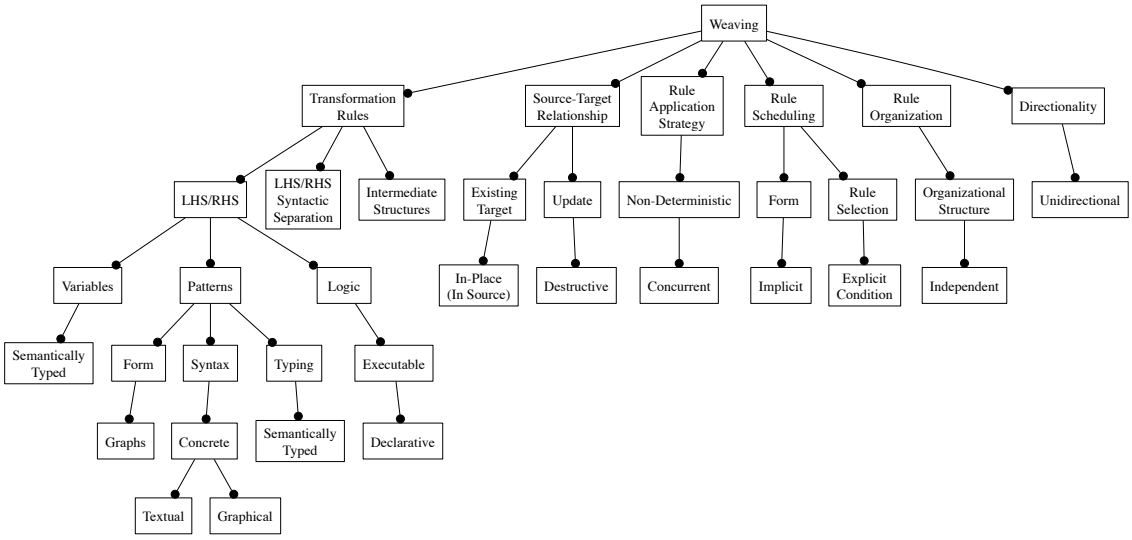


Figure 4.19: Classification of the Weaving model transformation

- **Transformation Rules**: **Weaving** relies on SPARQL `CONSTRUCT` queries and is based on **Fragments**, their results can be thought of as intermediate structures, the **Fragments** they use are parameterized semantically-typed graph patterns whose logic is declarative and they separate left-hand side **Pattern** from right-hand side **Blueprint** syntactically;
- **Source-Target Relationship**: the result of the SPARQL `CONSTRUCT` query is merged with the base workflow in what amounts to an in-place destructive update;
- **Rule Application Strategy**: the graph transformation defined by a **Fragment** is applied concurrently wherever the **Pattern** matches the base workflow;
- **Rule Scheduling**: scheduling of rules is performed implicitly by the SPARQL engine according to the explicit conditions the **Pattern** defines;
- **Rule Organization**: **Fragments** are stored in and retrieved from the knowledge base independently from the **Conceptual Workflow** being mapped; and
- **Directionality**: though it is possible for any **Weaving** application to build a **Fragment** undoing its effects, the **Weaving** mechanism itself is unidirectional.

4.3 Tools

The following two tools, **Merging** and **Erasing**, are provided to the user to help reach the desired result after, respectively, link-bound and node-bound **Weaving** (*cf.* Section 4.2.6).

In some cases, the result of **Weaving** will be directly desirable, therefore those tools are not automatically used and are not considered to be part of **Weaving** per se. Instead, they are provided for the user to exploit at will.

4.3.1 Merging

Merging is a tool melding two **Conceptual Workflows** into one. What it does in practice, after checking that the two input **Conceptual Workflows** are of the same type (*i.e.* two **Conceptual Functions**, two **Conceptual Inputs** or two **Conceptual Outputs**), is removing all the elements contained in one of them (*i.e.* the second argument), all links linking to or from it as well as its annotations, adding them progressively to the other (*i.e.* the first argument) and finally removing the now completely empty **Conceptual Workflow**.

Resulting **Conceptual Workflows** are named after the first argument. If the name of the second argument differs, then that information will be lost through **Merging**. That loss seems acceptable, since **Merging** is meant to be used on virtually identical **Conceptual Workflows**.

Merging becomes especially useful after link-bound **Weaving**, when multiple paths generate as many new elements, even though only one was desired. To illustrate that use, let us reconsider the example given in Section 4.2.6.2: because the base workflow had two possible paths through the `Alignment` step (one per **Conceptual Input**), link-bound **Weaving** produces not one `Registration` and one `Transformation` step, but two of each.

By using **Merging** on the twin `Registration` steps and the twin `Transformation` ones, the user can, with minimal hassle, achieve the desired result, shown on Figure 4.20, without modifying the **Fragment** in a way that would restrict it (*e.g.* putting two links targeting the `Alignment` step to account for the two paths present in the base workflow).

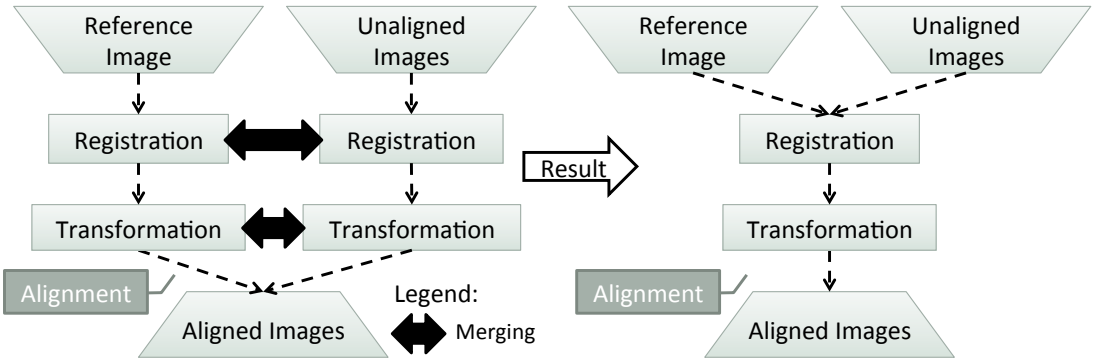


Figure 4.20: Merging Example

4.3.2 Erasing

Erasing is a tool that not only removes a **Conceptual Workflow**, but reassigns the elements associated with it (*i.e.* **Conceptual Links**, **Annotations** and sub-workflows) so as not to lose them. It is a tool meant to help the user get rid of potentially undesired levels of encapsulation created by node-bound **Weaving** and it is restricted to **Conceptual Workflows** embedding no **Abstract Elements** to ensure that links between high-level concepts and low-level elements are not needlessly destroyed.

The process is a bit more involved than that of **Merging**, because the way elements are reassigned depends on their type.

4.3.2.1 Conceptual Links

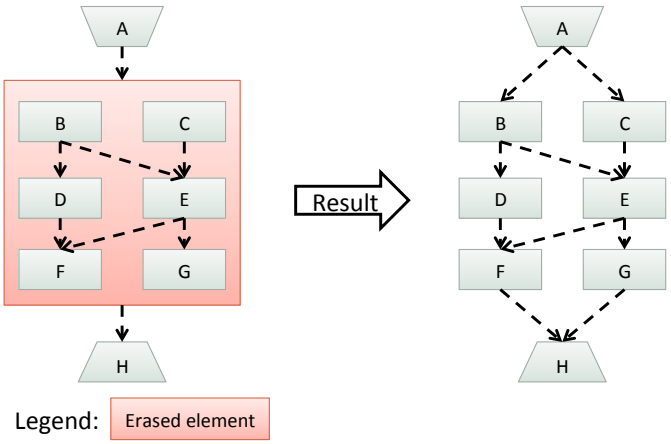


Figure 4.21: Erasing - Links

Conceptual Links targeting (resp. coming from) the erased **Conceptual Workflow** are reassigned to the **Conceptual Functions** it contains that have no incoming (resp. outgoing) links, since it makes them *de facto* entry (resp. exit) points of the workflow, as illustrated by Figure 4.21.

In cases where there are no suitable **Conceptual Functions** in the erased one, the incoming links are combined with the outgoing links, in such a way that there is a link between each predecessor and each successor of the erased **Conceptual Workflow**, as illustrated by Figure 4.22.

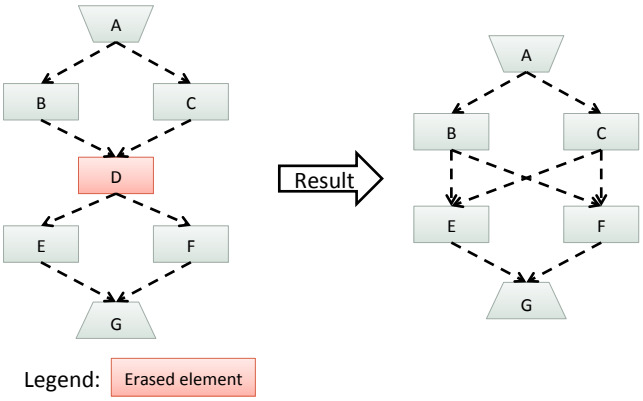
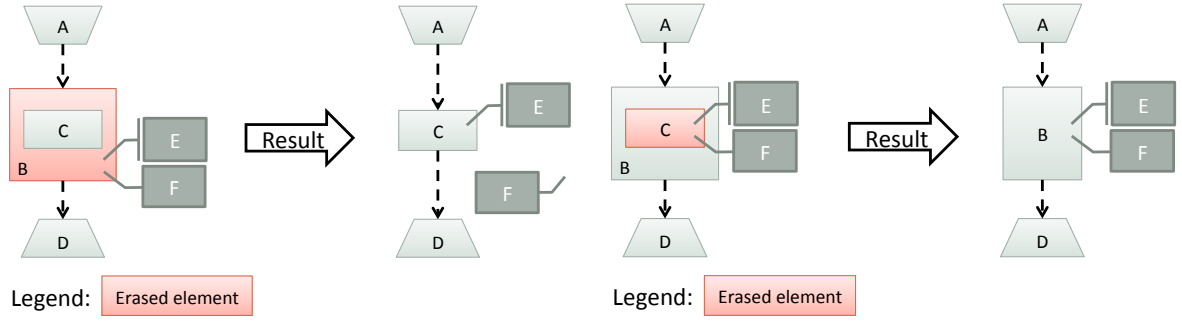


Figure 4.22: Erasing - Links Bypass

4.3.2.2 Annotations

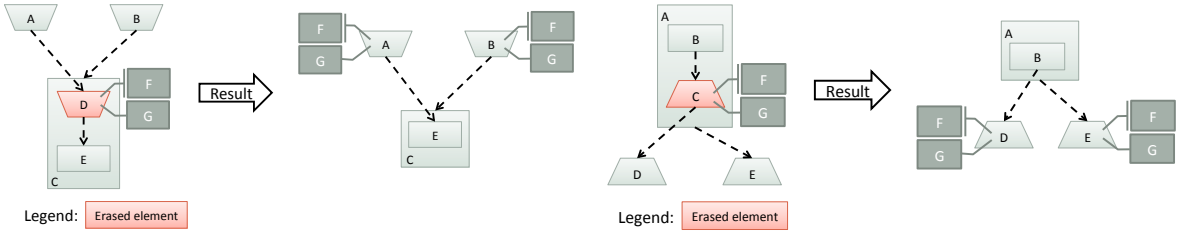
When it comes to reassigning **Annotations**, there are 4 cases, depending on the nature of the erased **Conceptual Workflow** and the **Meaning** of the **Annotations**:

- for **Conceptual Functions** with sub-**Conceptual Functions**, the **Requirements** are re-assigned to those sub-functions and the **Specifications** to the parent, as illustrated by Figure 4.23a;
- for **Conceptual Functions** without sub-**Conceptual Functions**, all **Annotations** are re-assigned to the parent, as illustrated by Figure 4.23b;
- for **Conceptual Inputs**, **Annotations** are re-assigned to all compatible predecessors, as illustrated by Figure 4.23c; and
- for **Conceptual Outputs**, **Annotations** are re-assigned to all compatible successors, as illustrated by Figure 4.23d.



(a) Conceptual Functions with Sub-functions

(b) Conceptual Functions without Sub-functions



(c) Conceptual Inputs

(d) Conceptual Outputs

Figure 4.23: Erasing - Annotations

4.3.2.3 Sub-workflows

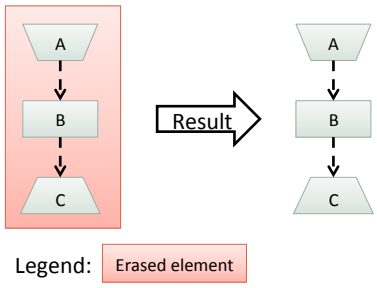


Figure 4.24: Erasing - Sub-workflows

Sub-workflows of the erased **Conceptual Workflow** become sub-workflows of its parent, as illustrated by Figure 4.24.

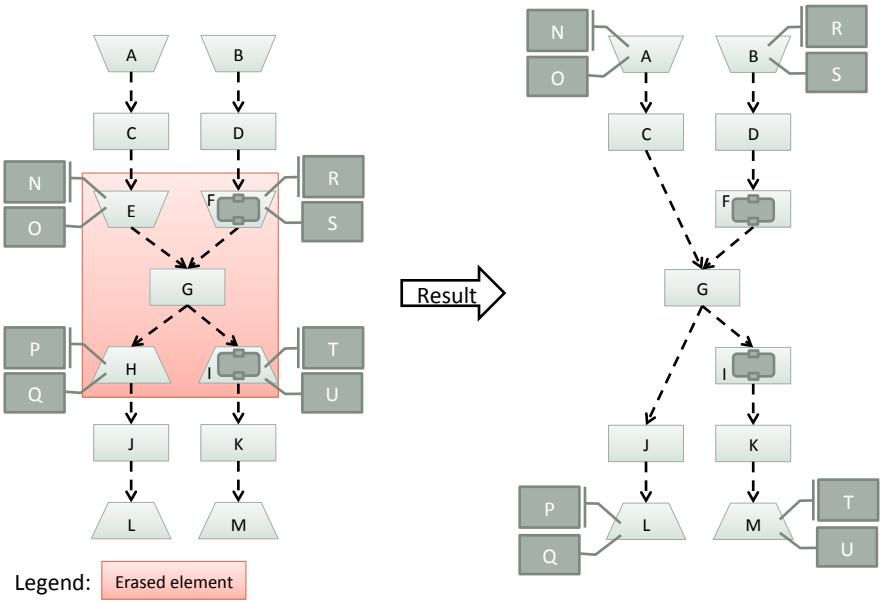


Figure 4.25: Erasing - Link Constraints

If the sub-workflow is a **Conceptual Input** (resp. **Conceptual Output**) with incoming (resp. outgoing) links that come from (resp. target) a sub-workflow of its new parent (*i.e.* the parent of the erased **Conceptual Workflow**), then the reassignment violates the constraint imposed on **Conceptual Links**, *cf.* Section 3.1.4. There are two cases, both illustrated by Figure 4.25:

- if the **Conceptual Input** (resp. **Conceptual Output**) embeds no **Abstract Elements**, then it is itself recursively erased;

- otherwise, it is transformed into a **Conceptual Input** and its **Annotations** (both **Requirements** and **Specifications**) are transferred to all compatible predecessors (resp. successors).

4.3.2.4 Example

Back to our running example, after node-bound **Weaving**, by using **Erasing** on the leftover `Alignment` step, the user can achieve the same result as with link-bound **Weaving** then **Merging**, as shown on Figure 4.26.

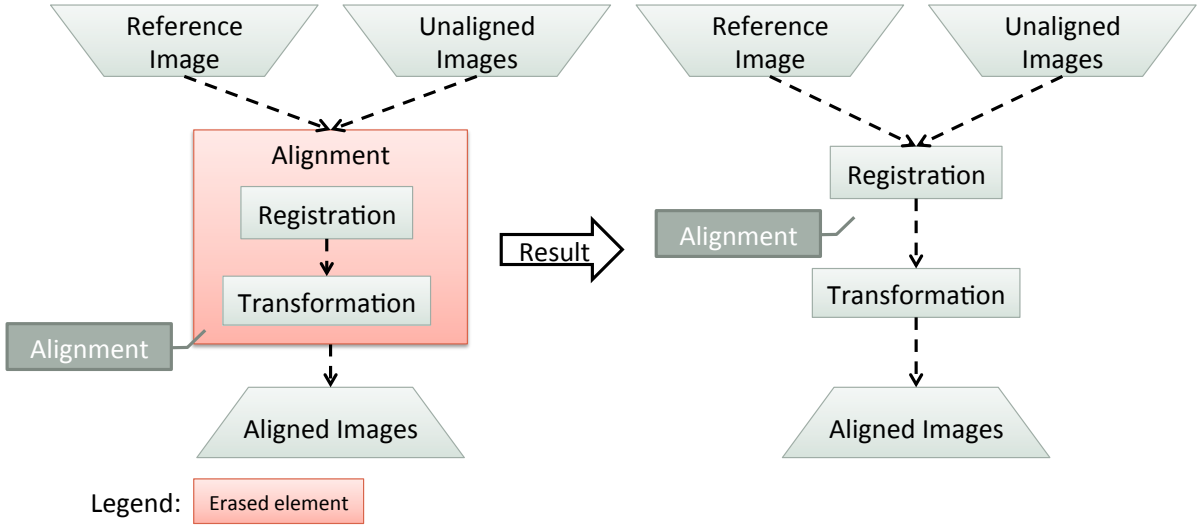


Figure 4.26: Erasing - Example

4.4 Discovery

The usefulness of **Weaving** is limited by the quantity and quality of available **Fragments** and by the ability of end users to find the **Fragments** which best fit their needs. The user community can be thought of as responsible for the quality of the **Fragments** present in their knowledge base, but even plentiful relevant **Fragments** will be of little use if they are not found.

One of the main reasons why **Mapping** cannot be fully automated is that finding a relevant **Fragment** is not guaranteed: there might be no such **Fragment** in the knowledge base or it might be annotated inconsistently with the base **Conceptual Workflow**.

However, it is possible to help users find candidate **Fragments** based on **Annotations**, in cases where matches exist. For the system to suggest **Fragments** in as many cases as possible, it is also important to broaden results, *i.e.* to suggest **Fragments** which are less-than-perfect matches for the situation. Though those **Fragments** might not perform all desired **Functions** and/or fulfill all **Concerns**, they might give part of the answer or serve for the user as clues as to what can be done.

4.4.1 Process

The process of helping users find **Fragments** of varied relevance is called **Discovery** and is done in four steps, as is clear from the modeling of the process in our own **Conceptual Workflow Model**, shown on Figure 4.27:

- **Selection**: the user selects a **Conceptual Workflow** with at least one **Requirement**;
- **Matching**: the **Requirements** of that **Conceptual Workflow** are compared to that of **Fragments** contained in the knowledge base, as explained in Section 4.4.2;

- **Ranking**: the system orders the matches found at the previous step by order of probable relevance, as explained in Section 4.4.3; and
- **Choice**: the ordered list of candidates is presented to users for them to choose the **Fragment** that will be used as input for **Weaving**.

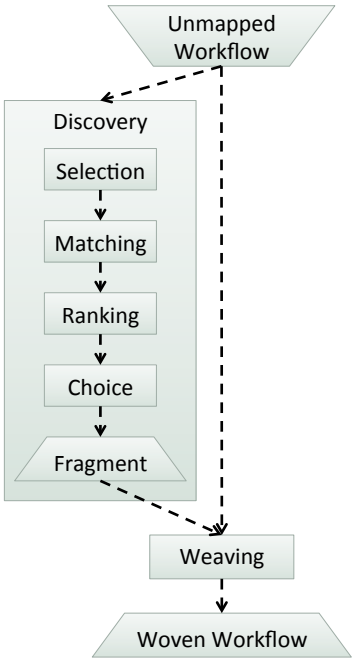


Figure 4.27: Discovery Process

4.4.2 Matching

Before **Matching**, the user selects a **Conceptual Workflow** with one or more **Requirements**. The objective of **Matching** is to find **Fragments** in the knowledge base with **Annotations** corresponding to those **Requirements**.

Trying to find all **Requirements** simultaneously would miss all **partial matches** (*i.e.* **Fragments** bearing **Annotations** matching only part of the target **Requirements**) and looking for all possible combinations of **Requirements** would not scale well with the number of **Requirements**. The best option is thus to look for **Fragments** matching each **Requirement** separately and getting rid of redundancies.

However, it would make little sense to compare a **Requirement** with all **Annotations** indiscriminately. Indeed, the **Role** of **Annotations** matters as well as their location inside the **Fragments**:

- a **Requirement** present...
  - ...only in the **Pattern** represents an objective of the **Fragment**, *i.e.* a **Function** it performs or a **Concern** it fulfills;
  - ...in both the **Pattern** and the **Blueprint** represents a context in which the **Fragment** is relevant and that it preserves;
  - ...only in the **Blueprint** represents a need for further **Mapping**; and

- a **Specification** present...
  - ...only in the **Pattern** represents a context in which the **Fragment** is relevant and that it modifies;
  - ...in both the **Pattern** and the **Blueprint** represents a context in which the **Fragment** is relevant and that it preserves;
  - ...only in the **Blueprint** represents a result of the **Fragment**, *i.e.* a **Function** it performs or a **Concern** it fulfills.

Therefore, **Matching** limits comparison to the **Requirements** present in **Patterns** and the **Specifications** present in **Blueprints**, so as to focus on the objectives and results of **Fragments**.

4.4.2.1 Match Quality

When comparing **Annotations**, it is not enough to determine whether they are equal, *i.e.* share the same **Type**. Indeed, doing only that would make **Matching** only more restrictive than keyword-based search, since keywords would be limited to terms defined in an ontology. To leverage semantic knowledge, it is necessary to take into account class hierarchies, at the very least. Obviously, the further apart classes are in a taxonomy graph, the less similar the two classes are, but direction counts just as much as distance.

Indeed, whenever a `Sub rdfs:subClassOf Super` triple is asserted, since all instances of `Sub` are instances of `Super` but instances of `Super` are not necessarily instances of `Sub`, it induces the following two relationships between the two classes: `Sub` is **narrower** than `Super` and `Super` is **broader** than `Sub`.

For instance, in OntoVIP, the ontology of the Virtual Imaging Platform (VIP), *cf.* Section 5.2.1, `affine-registration` is **broader** than `rigid-registration` and **narrower** than `registration`, as can be seen on the very small excerpt shown on Figure 4.28.

For two given **Annotations** `Candidate` (the potential match whose quality is evaluated) and `Target` (the reference to match), we define three distinct match qualities (all other cases are considered to be no match at all):

- **Exact match** applies when `Candidate` and `Target` share the **same type**;
- **Narrower match** applies when the `Candidate` is a **subtype** of the `Target`; and
- **Broader match** applies when the `Candidate` is a **supertype** of the `Target`.

The case for **narrower matches** seems pretty straightforward, at first glance: if a user is looking for an affine rigid registration process, they will likely want both mono and multi modality versions to come up in their search. In fact, the need to find **narrower matches** is so ubiquitous that the inference rule attached to the basic property `rdfs:subClassOf` ensures they are found. For instance, if:

- `Proc rdf:type vip:mono-modality-affine-rigid-registration` is asserted and
- inferences are run against the VIP ontology, then
- `Proc rdf:type vip:affine-rigid-registration` will be inferred; and
- looking for instances `vip:affine-rigid-registration` will find `Proc`.

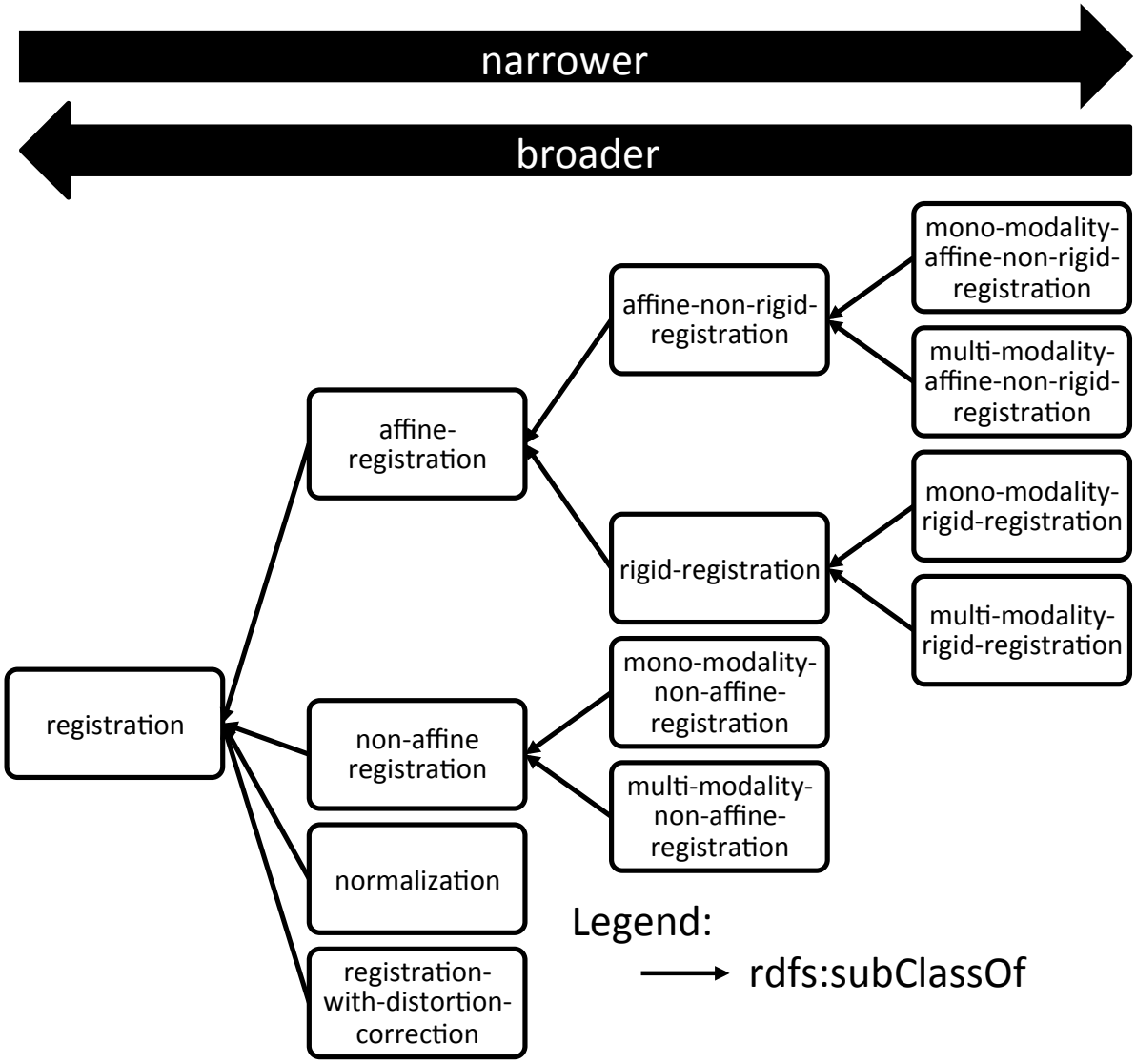


Figure 4.28: VIP Ontology (excerpt) - Registration Processes Taxonomy

The utility of **broader matches** is less obvious. They are deemed moderatly relevant and of a lower quality than both **exact** and **narrower** matches, for two reasons:

- the user might over-specify, *i.e.* look for a supertype of what they really need, *e.g.* look for a rigid registration algorithm when a non-rigid one would do just as well in their situation; and

- there might be valuable insight in workflows that are slightly too broad in scope for the work at hand.

However, neither reason holds up when distance increases between candidate and target: the broader the candidate is compared to the target, the less likely it is that it will be of any interest for the user. For instance, if the user is looking for an affine registration process, other types of registration processes might be of some value, but climb higher in the taxonomy and you find `dataset-processing`, which is obviously way too broad to be of any use.

Unfortunately, all `rdfs:subClassOf` relationships are equal, therefore it is generally impossible to tell automatically whether a superclass is too broad to still be relevant. To avoid burdening the user with completely irrelevant **Fragments** and yet attempt not to miss out on all potentially interesting **broader matches**, we have decided to limit the search to direct broader matches, *i.e.* candidates whose type is the direct supertype of the target's, instead of recursively.

4.4.2.2 Matching Query

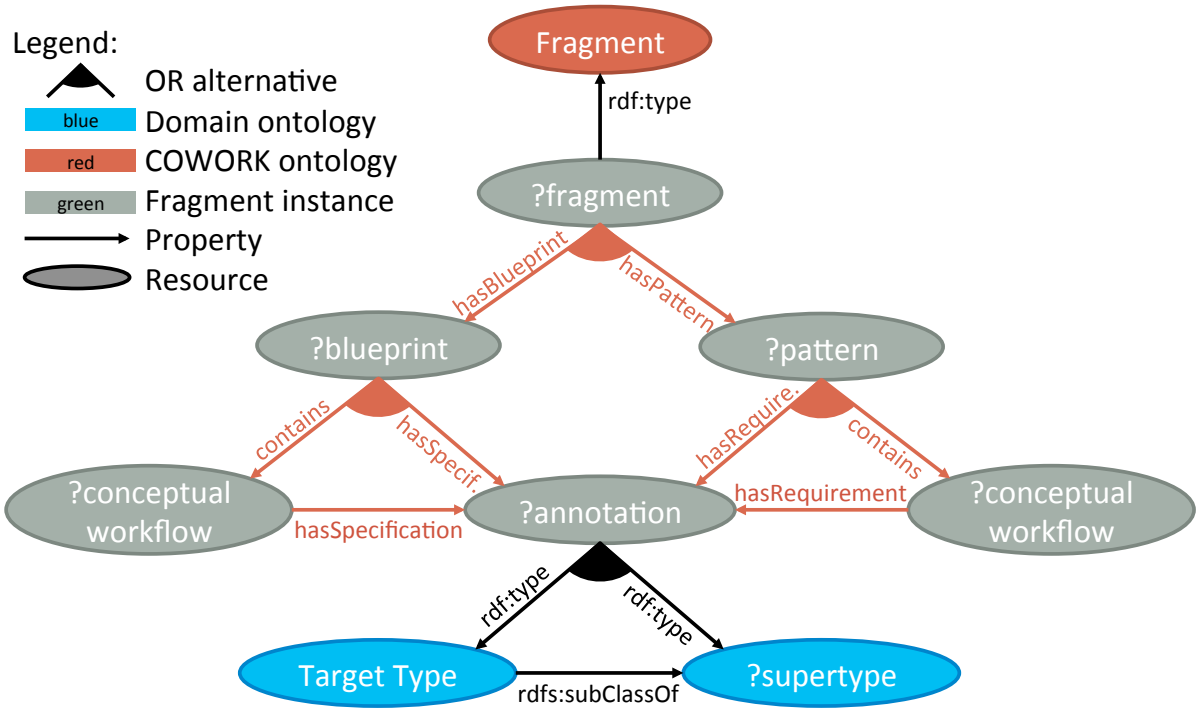


Figure 4.29: Matching Query

The most efficient way to identify matching **Fragments** in the knowledge base is through a SPARQL `SELECT` query. The query must look for many alternative graph patterns simultaneously. Namely, as illustrated by Figure 4.29, it must look for:

- either a **Requirement** in a **Pattern** or a **Specification** in a **Blueprint**;
- either an **Annotation** born directly by the **Pattern/Blueprint** or born by a **Conceptual Workflow** contained therein; and
- an **Annotation** of either the target type (*i.e.* an **exact** or **narrower** match) or a supertype of the target type (*i.e.* a **broader** match).

In SPARQL, alternative graph patterns are declared with the keyword `UNION`. Whenever two alternatives are given through that keyword, the query engine will try to match both separately and return all results matching either or both. Listing 4.3 shows the basic SPARQL query for **Matching**.

Listing 4.3: Matching Query (`T` = target type)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cowork: <http://www.i3s.unice.fr/~cerezo/cowork/latest/cowork.rdfs#>

SELECT ?fragment  WHERE {
  ?fragment rdf:type cowork:Fragment .

  {
    ?fragment cowork:hasBlueprint ?blueprint .
    {
      ?blueprint cowork:hasSpecification ?annotation .
    }
    UNION
    {
      ?blueprint cowork:contains ?conceptualworkflow .
      ?conceptualworkflow cowork:hasSpecification ?annotation .
    }
  }
  UNION
  {
    ?fragment cowork:hasPattern ?pattern .
    {
      ?pattern cowork:hasRequirement ?annotation .
    }
    UNION
    {
        ?pattern cowork:contains ?conceptualworkflow .
        ?conceptualworkflow cowork:hasRequirement ?annotation .
    }
  }

  {
   ?annotation rdf:type T .
  }
  UNION
  {
    ?annotation rdf:type ?supertype .
    T rdfs:subClassOf ?supertype .
  }
}
```

4.4.3 Ranking

After retrieving all **Fragments** matching any of the **Requirements** of the selected **Conceptual Workflow**, those **Fragments** must be ranked by order or probable relevance for the user. In turn, the **Annotations** and elements of each **Fragment** are compared to that of the selected **Conceptual Workflow** in order to evaluate that relevance, following four principles described in the following Section 4.4.3.1.

The algorithm itself differs depending on the type of **Conceptual Workflow** selected: Section 4.4.3.3 details the **Ranking** algorithm for **Conceptual Inputs**, Section 4.4.3.4 for **Conceptual Outputs** and Section 4.4.3.5 for **Conceptual Functions**.

4.4.3.1 Ranking Principles

Prioritize match quality. For the reasons given in Section 4.4.2.1, all other things being equal, an **exact** match should rank higher than both **narrower** matches and **broader** matches; and a **narrower** match should rank higher than **broader** matches. For instance, when looking for affine registration processes, those annotated exactly likewise should come up first, then those annotated more specifically and non-affine registration processes should come last.

Penalize partial matches. All other things being equal, the more **Requirements** a **Fragment** matches, the higher it should rank. For instance, when looking for a parallel registration process, sequential registration processes should rank lower than parallel ones, since they match only the **Function** and not the parallelism **Concern**.

Prioritize Functions over Concerns. This is only relevant for **Conceptual Functions**, since **Conceptual Inputs** and **Conceptual Outputs** only bear **Datasets** (*cf.* Section 3.3.4). It makes more sense to look first for **Fragments** performing the desired **Functions** and worry later about the non-functional **Concerns**, rather than the opposite. For instance, when looking for a parallel registration process, a sequential registration process would be more relevant to the search than a parallel gene sequence alignment process, even though both fulfill the parallelism **Concern**.

Penalize extras. If the selected **Conceptual Workflow** is a **Conceptual Function**, then the more extra **Functions** (*i.e.* **Functions** matching no **Requirements**) a **Fragment** bears, the lower it should rank. For instance, when looking for a registration process, a process performing both registration and denoising should rank lower than a process performing only the desired **Function**.

If the selected **Conceptual Workflow** is a **Conceptual Input** (resp. **Conceptual Output**), then **Fragments** that would generate extra **Conceptual Inputs** (resp. **Conceptual Outputs**) through **Weaving** (*i.e.* **Fragments** with **Conceptual Inputs** (resp. **Conceptual Outputs**) featured only in their **Blueprint**), should rank lower than **Fragments** that would not.

4.4.3.2 Scoring

All three **Ranking** algorithms presented thereafter rely on the following $score$ function:

Algorithm 1 Scoring Annotations or Elements

```
1:  function SCORE(t : Annotation, c : Annotation)
2:      if type(t) = type(c) then                          ▷ If t and c are of the same type,
3:          return K_EM                                    ▷ then return the exact match constant;
4:      else if type(t) ∈ superclasses(type(c)) then       ▷ else if c's type derives from t's,
5:          return K_NM                                    ▷ then return the narrower match constant;
6:      else if type(c) = superclass(type(t)) then         ▷ else if t's type is a subclass of c's,
7:          return K_BM                                    ▷ then return the broader match constant;
8:      else
9:          return 0.0                                     ▷ else return zero (no match).
10:     end if
11: end function
12: function SCORE(t : Element, c : Element)
13:     return max_{a_t∈Γ(t), a_c∈Γ(c)}(score(a_t, a_c))    ▷ Return maximum score.
14: end function
```

As detailed in Algorithm 1, when applied on two **Annotations**, it evaluates and returns their match quality, as described in Section 4.4.2.1 and when applied on two elements, such as **Conceptual Functions**, it returns the maximum score between their respective **Annotations**.

It uses the following sub-functions:

- $type(a : Annotation)$ returns a's **Type**;
- $superclasses(c : Class)$ returns the set of all superclasses of c; and
- $superclass(c : Class)$ returns the direct superclass of c.

It also uses three constants K_{EM}, K_{NM} and K_{BM} defining the default scores for **exact**, **narrower** and **broader** matches, respectively. Though it is clear, as explained in Section 4.4.2.1, that $K_{EM} > K_{NM} > K_{BM} > 0$, the best value for each constant depends on the content of the knowledge base and the preferences of the user. It is thus better not to freeze them and instead give them default values which can be tweaked by users if they so wish.

4.4.3.3 Ranking for Conceptual Inputs

Algorithm 2 Ranking candidates for a Conceptual Input

```
1:  function DISCOVER(ci : ConceptualInput)
2:      result : Dictionary(Fragment, Float)          ▷ Initialize result list.
3:      for all r ∈ Γ_R(ci) do                        ▷ ∀r requirement of ci,
4:          for all m ∈ match(r) do                   ▷ ∀m candidate match for r:
5:              w ← max_{a∈Γ(m)}(score(a, r))         ▷ Compute best Matching score.
6:              if m ∈ keys(result) then              ▷ If m is already in the result list,
7:                  result[m] ← result[m] + w         ▷ then update its weight;
8:              else
9:                  result[m] ← w                     ▷ else add m to the list with default weight.
10:             end if
11:         end for
12:     end for
13:     for all (m, w) ∈ result do                    ▷ ∀(m, w) match and associated weight:
14:         result[m] ← result[m] / |Γ_R(ci)|         ▷ Penalize partial matches.
15:         result[m] ← result[m] / (1+|inputs(m)|)   ▷ Penalize extra inputs.
16:     end for
17:     return sort(result)                           ▷ Sort by decreasing weight.
18: end function
```

The $discover$ function detailed in Algorithm 2 finds candidate **Fragments** for a selected **Conceptual Input** and ranks them. It uses the $score$ function defined in Section 4.4.3.2 and the following sub-functions:

- $match(r : Requirement)$ uses the SPARQL query detailed in Listing 4.3, in Section 4.4.2, to fetch all candidate matches for the **Requirement** r; and
- $inputs(f : Fragment)$ returns the set of **Conceptual Inputs** contained in f's **Blueprint**.

4.4.3.4 Ranking for Conceptual Outputs

The $discover$ function detailed in Algorithm 3 finds candidate **Fragments** for a selected **Conceptual Output** and ranks them. It uses the same functions as Algorithm 2, with the addition of $outputs(f : Fragment)$, which returns the set of **Conceptual Outputs** contained in f's **Blueprint**.

Algorithm 3 Ranking candidates for a Conceptual Output

function DISCOVER($co : ConceptualOutput$)
  $result : Dictionary(Fragment, Float)$ ▷ Initialize result list.
  for all $r \in \Gamma_R(co)$ **do** ▷ $\forall r$ requirement of co,
    for all $m \in match(r)$ **do** ▷ $\forall m$ candidate match for r:
      $w \leftarrow max_{a \in annot(m)}(score(a, r))$ ▷ Compute best Matching score.
      if $m \in keys(result)$ **then** ▷ If m is already in the result list,
        $result[m] \leftarrow result[m] + w$ ▷ then update its weight;
      else
        $result[m] \leftarrow w$ ▷ else add m to the list with default weight.
      end if
    end for
  end for
  for all $(m, w) \in result$ **do** ▷ $\forall (m, w)$ match and associated weight:
    $result[m] \leftarrow \frac{result[m]}{|\Gamma_R(co)|}$ ▷ Penalize partial matches.
    $result[m] \leftarrow \frac{result[m]}{|outputs(m)|+1}$ ▷ Penalize extra outputs.
  end for
  return $sort(result)$ ▷ Sort by decreasing weight.
end function

4.4.3.5 Ranking for Conceptual Functions

The $discover$ function detailed in Algorithm 4 finds candidate **Fragments** for a selected **Conceptual Function** and ranks them. It uses the same functions as Algorithm 2, with the addition of $meaning(a : Annotation)$, which returns the **Meaning** of a, *i.e.* either **Function**, **Concern** or **Dataset**. It also uses a constant K_F which is a factor characterizing how much **Functions** should be prioritized over **Concerns**.

As with K_{EM}, K_{NM} and K_{BM}, the best value for the constant K_F depends on the content of the knowledge base and the preferences of the user and it is best not to give it a default value which can be tweaked by users if they so wish.

4.5 Composition

Once all **Requirements** are fulfilled, it is very likely the resulting workflow will be incomplete: there most probably will be disconnected **Activities** among those woven or inserted into the **Conceptual Workflow**. For the workflow to become a viable **Intermediate Representation**, those **Activities** must be composed, *i.e.* all **Input Ports** must be bound to **Output Ports**.

It is at this level that most technical issues have to be dealt with, most notably format conversion. Indeed, so far the focus has been purposely put on high-level concepts, leaving out technicalities so as to emphasize the scientific experiment over its implementation.

However, to transform the **Conceptual Workflow** into a fully executable abstract workflow still requires tackling all those technical issues.

Like **Mapping**, **Composition** cannot be fully automated, since that would assume that every required piece of know-how is available and that it is accurately annotated. There is most often a gap between the "*physical layer*", where the enactor handles files formats, error codes and transfer protocols, and the "*semantic layer*", where **Semantic Annotations** lie.

Algorithm 4 Ranking candidates for a Conceptual Function

function DISCOVER($cf : ConceptualFunction$)
  if $\exists c / cf \prec c$ **then** ▷ If there are sub-workflows,
    $result : Dictionary(ConceptualWorkflow, Object)$ ▷ then initialize list
    for all $c / cf \prec c$ **do** ▷ and $\forall c$ sub-workflow
      $result[c] \leftarrow discover(c)$ ▷ propagate to c recursively.
    end for
    return $result$ ▷ Return results of recursive propagation.
  else
    $result : Dictionary(Element, Float)$ ▷ Initialize result list.
    for all $r \in \Gamma_R(cf)$ **do** ▷ $\forall r$ requirement of cf,
      for all $m \in match(r)$ **do** ▷ $\forall m$ candidate match for r:
        $w \leftarrow max_{a \in \Gamma(m)}(score(a, r))$ ▷ Compute best Matching score.
        if $meaning(r) = Function$ **then** ▷ If r is a Function,
          $w' \leftarrow K_F * w$ ▷ then increase default weight by a factor K_F;
        else ▷ else (*i.e.* if r is a Concern)
          $w' \leftarrow w$ ▷ leave default weight as-is.
        end if
        if $m \in keys(result)$ **then** ▷ If m is already in the result list,
          $result[m] \leftarrow result[m] + w'$ ▷ then update its weight;
        else
          $result[m] \leftarrow w'$ ▷ else add m to the list with default weight.
        end if
      end for
    end for
    for all $(m, w) \in result$ **do** ▷ $\forall (m, w)$ match and associated weight:
      $result[m] \leftarrow \frac{result[m]}{K_F * |\Gamma_R(cf) \cap \Omega_F| + |\Gamma_R(cf) \cap \Omega_C|}$ ▷ Penalize partial matches.
      $result[m] \leftarrow \frac{result[m]}{1 + |\Gamma_S(m) \setminus \Gamma_R(cf)|}$ ▷ Penalize extra Functions.
    end for
    return $sort(result)$ ▷ Sort by decreasing weight.
  end if
end function

Automation can be improved by extending either or both layers, *e.g.* augmenting basic types with meta-data and/or extending ontologies with technical notions as they arise. But, as with the rest of the **Mapping** process, there is a trade-off between the scope of the system and the level of automation that can be provided.

Without a closed-world assumption, user input is needed to sort out non-modeled **Requirements** and fill in knowledge and know-how that was never captured. Nonetheless, workflow design can be assisted in cases where **Annotations** provide enough information to detect and/or solve technical issues.

We leverage **Semantic Annotations** in four different ways highlighted on Figure 4.30:

- to suggest **Data Links** between compatible **Ports** in the workflow, as explained in Section 4.5.1;
- to suggest **Activities** from the knowledge base that could produce the type of data required by an existing **Activity**, as explained in Section 4.5.2;
- to conversely suggest **Activities** from the knowledge base that could consume the type of data produced by an existing **Activity**, as explained in Section 4.5.3; and
- to suggest **Activities** (or chains thereof) from the knowledge base that could fix a mismatched **Data Link** in the workflow, as explained in Section 4.5.4.

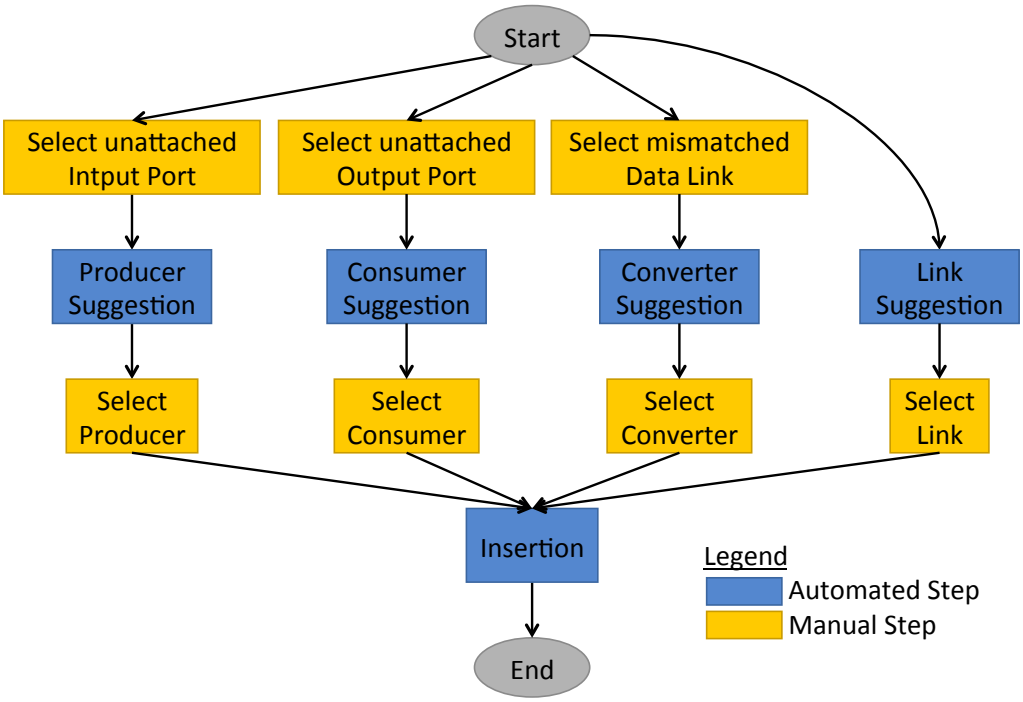


Figure 4.30: Composition Process

4.5.1 Link Suggestion

When a **Port** is not bound by any **Data Link**, it is **unattached**. While unattached **Output Ports** are not necessarily a problem (they might be by-products of an **Activity** that are irrelevant to the simulation at hand), all unattached **Input Ports** induce unreachable parts in the workflow.

Indeed, if no data is ever directed to an **Activity**, it will never be executed and neither will be those depending on its execution (through **Order Links**) or its products (through **Data Links**).

The link suggestion algorithm examines all unattached **Input Ports** in the **Conceptual Workflow** and suggests **Data Links** between them and any **compatible Output Port** already present in the workflow.

An **Output Port** is **compatible** with an **Input Port** if and only if (i) the associated **Data Link** would preserve the order defined by **Conceptual Links** and (ii) their **Annotations** match, *i.e.* their score, as defined in Section 4.4.3.2, is not null:

Ports Compatibility

$$
\begin{array}{rrcl}
 & \forall c_S = (V_S, E_S) & \in & H \\
 & \forall c_T = (V_T, E_T) & \in & H \\
 & \forall o & \in & L_{OP} \cap V_S \\
 & \forall i & \in & L_{IP} \cap V_T \\
 & o \rightarrowtail i & \textit{i.e.} & o \text{ is compatible with } i \\
\text{iff:} & (c_S = c_T & \text{or} & c_S \rightsquigarrow c_T) \\
\text{and} & score(o, i) & > & 0
\end{array}
$$

Subsequently, suggesting links to the user for a given **Conceptual Workflow** consists in listing all its unattached **Input Ports** and finding all compatible **Output Ports**.

It is done through two functions:

- $predecessing(cw : ConceptualWorkflow)$, detailed in Algorithm 5, finds all **Output Ports** predecessing the **Conceptual Workflow** cw; and
- $link(cw : ConceptualWorkflow)$, detailed in Algorithm 6, checks whether predecessing **Output Ports** are compatible and suggests them if they are.

Algorithm 5 Finding predecessing Output Ports

```
1:  function PREDECESSING(cw : ConceptualWorkflow)
2:      result : List(OutputPort)                          ▷ Initialize result list.
3:      for all o ∈ L_OP ∩ immediateOutputs(cw) do   ▷ ∀o Output Port immediately in cw:
4:          result ↞ o                                     ▷ Append o to the list.
5:      end for
6:      for all cl ∈ incomingLinks(cw) do                  ▷ ∀cl Conceptual Link targeting cw:
7:          result ← result ∪ predecessing(source(cl))     ▷ Propagate to cl's source.
8:      end for
9:      if ∃c_P/c_P ≼ cw then                              ▷ If cw has an immediate parent c_P,
10:         result ← result ∪ predecessing(c_P)            ▷ then propagate to c_P.
11:     end if
12:     return result                                      ▷ Return all predecessing Output Ports.
13: end function
```

The $predecessing$ function uses the following sub-functions:

- $immediateOutputs(cw : ConceptualWorkflow)$ returns the set of **Conceptual Outputs** and **Output Ports** directly contained in cw (by opposition to those possibly contained in sub-workflows); and
- $incomingLinks(cw : ConceptualWorkflow)$ returns the set of **Conceptual Links** targeting cw.

Algorithm 6 Suggesting Links

```
1:  function LINK(cw : ConceptualWorkflow)
2:      result : List(DataLink)                          ▷ Initialize result list.
3:      for all i ∈ L_IP ∩ inputs(cw) do                 ▷ ∀i Input Port in cw:
4:          c_P ← immediateParent(activity(i))           ▷ Let c_P be the immediate parent of i
5:          for all o ∈ predecessing(c_P) do             ▷ ∀o predecessing Output Port:
6:              if score(i, o) > 0 then                  ▷ If i's and o's Annotations match,
7:                  result ↞ DataLink(o, i)              ▷ append Data Link o → i to the result list.
8:              end if
9:          end for
10:     end for
11:     return result                                    ▷ Return suggested links.
12: end function
```

The $link$ function uses the following sub-functions:

- $inputs(cw : ConceptualWorkflow)$ returns the set of **Conceptual Inputs** and **Input Ports** contained in cw at any depth;
- $immediateParent(a : Activity)$ returns the **Conceptual Function** in which a is directly embedded (by opposition to the parent **Conceptual Workflows** of the immediate parent); and
- $activity(p : Port)$ returns the **Activity** p belongs to.

4.5.2 Producer Suggestion

If the algorithm of the previous Section 4.5.1 does not suggest any **Data Link** for a given **unattached Input Port**, or if none of the suggestions fit the needs of the user, there might be an **Activity** in the knowledge base which could be a **producer**, *i.e.* an **Activity** with an **Output Port** whose **Specifications** matches that of the unattached **Input Port**.

Listing 4.4 gives the SPARQL query which looks for all **Activities** producing type `T`: *i.e.* all **Activities** having an **Output Port** bearing a **Specification** which is an exact, narrower or broader match for `T`. In the former case, the **Specification** will be of type `T` directly or through inference, whereas, in the latter, the it will be of a direct supertype of `T`.

Listing 4.4: Producer Search Query (`T` = target type)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cowork: <http://www.i3s.unice.fr/~cerezo/cowork/latest/cowork.rdfs#>

SELECT ?activity WHERE {
  ?activity rdf:type cowork:Activity .

  ?activity cowork:hasOutput ?outputport .
  ?outputport cowork:hasSpecification ?dataset .
  {
    ?dataset rdf:type T .
  }
  UNION
  {
    ?dataset rdf:type ?supertype .
    T rdfs:subClassOf ?supertype .
  }
}
```

Much like candidate **Fragments** in the **Discovery** process (*cf.* Section 4.4), candidate producers must be ranked by order or probable relevance, according to the following principles:

- **prioritize match quality**: all other things being equal, **exact** matches should rank higher than **narrower** ones and both should rank higher than **broader** matches; and
- **penalize partial matches**: the more **Specifications** a producer matches at once, the higher it should rank.

Algorithm 7 Suggesting Producers

function PRODUCE($ip : InputPort$)
  $result : Dictionary(Activity, Float)$ ▷ Initialize result list.
  for all $s \in \Gamma_S(ip)$ **do** ▷ $\forall s$ Specification of ip:
    for all $a \in producers(s)$ **do** ▷ $\forall a$ Activity producing s:
      $w \leftarrow max_{s_a \in \Gamma_S(outputs(a))} score(s, s_a)$ ▷ Initialize weight.
      if $a \in keys(result)$ **then** ▷ If a is already in the result list,
        $result[a] \leftarrow result[a] + w$ ▷ then update its weight;
      else
        $result[a] \leftarrow w$ ▷ else add a to the list with weight w.
      end if
    end for
  end for
  for $(a, w) \in result$ **do** ▷ $\forall (a, w)$ Activity and associated weight:
    $result[a] \leftarrow \frac{result[a]}{|\Gamma_S(ip)|}$ ▷ Penalize partial matches.
  end for
  return $sort(result)$ ▷ Sort by decreasing weight.
end function

The function $produce$, detailed in Algorithm 7, looks for all candidate producers for each **Specification** of a selected **Input Port** and ranks them, using the $score$ function defined in Section 4.4.3.2 as well as the following sub-functions:

- $producers(d : Dataset)$ uses the SPARQL query of Listing 4.4 and retrieves all **Activities** from the knowledge base with an **Output Port** whose **Specifications** match d; and
- $outputs(a : Activity)$ returns the set of a's **Output Ports**.

4.5.3 Consumer Suggestion

Though unattached **Output Ports** are not necessarily problematic, it might occasionally help users to get suggestions for **Activities** that might be able to consume the data produced by an existing one. It is therefore interesting to suggest not only producers, but also **consumers**. The SPARQL query to look for **Activities** consuming a type `T` is given in Listing 4.5 and is very similar to the one given in Listing 4.4, which looks for producers.

Listing 4.5: Consumer Search Query (`T` = target type)

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cowork: <http://www.i3s.unice.fr/~cerezo/cowork/latest/cowork.rdfs#>

SELECT ?activity WHERE {
  ?activity rdf:type cowork:Activity .

  ?activity cowork:hasInput ?inputport .
  ?inputport cowork:hasSpecification ?dataset .
  {
    ?dataset rdf:type T .
  }
  UNION
  {
    ?dataset rdf:type ?supertype .
    T rdfs:subClassOf ?supertype .
  }
}
```

Algorithm 8 Suggesting Consumers

function CONSUME($op : OutputPort$)
  $result : Dictionary(Activity, Float)$ ▷ Initialize result list.
  for all $s \in \Gamma_S(op)$ **do** ▷ $\forall s$ Specification of op:
    for all $a \in consumers(s)$ **do** ▷ $\forall a$ Activity producing s:
      $w \leftarrow max_{s_a \in \Gamma_S(inputs(a))} score(s_a, s)$ ▷ Initialize weight.
      if $a \in keys(result)$ **then** ▷ If a is already in the result list,
        $result[a] \leftarrow result[a] + w$ ▷ then update its weight;
      else
        $result[a] \leftarrow w$ ▷ else add a to the list with weight w.
      end if
    end for
  end for
  for $(a, w) \in result$ **do** ▷ $\forall (a, w)$ Activity and associated weight:
    $result[a] \leftarrow \frac{result[a]}{|\Gamma_S(op)|}$ ▷ Penalize partial matches.
  end for
  return $sort(result)$ ▷ Sort by decreasing weight.
end function

The *consume* function looks for and then ranks candidate consumers for a selected **Output Port**, is detailed in Algorithm 8 and differs from the *produce* function in the following ways:

- it uses the *score* function defined in Section 4.4.3.2 **in reverse**, so that **broader** matches are ranked higher than **narrower** ones, because a **consumer** which supports a **broader** type should also support the **exact** one, whereas a **consumer** which supports a **narrower** type might not accept all data produced by the selected **Output Port**;

- it uses $consumers(d : Dataset)$ instead of $producers(d : Dataset)$, which uses the SPARQL query given in Listing 4.5 to retrieve all **Activities** from the knowledge base with an **Input Port** whose **Specifications** match d; and
- it uses $inputs(a : Activity)$ instead of $outputs(a : Activity)$, which returns the set of a's **Input Ports**.

4.5.4 Converter Suggestion

A **mismatch** characterizes a **Data Link** whose source and target are incompatible either at the physical level (if the basic types of source and target differ, *e.g.* list and string) or at the semantic level (if the target **Annotations** are neither exact nor narrower matches for the source ones). Since partial matches and broader matches are detected as mismatches, it is entirely possible for the aforementioned link suggestion algorithm to suggest mismatches. This is a trade-off in the effort to help the user compose the workflow. Indeed, mismatches will at least provide some clues as to some technical issues which must be tackled for the workflow to execute.

Further help can be provided to the user when mismatches are detected, regardless of how the mismatched **Data Links** were created (by the user, by the **Weaving** mechanism or by the link suggestion algorithm). If they lie at the physical level, the system can provide the user with a small collection of *ad-hoc* converters meant to tackle the most common format mismatches, such as `List2String` and `String2Integer`.

When mismatches are detected at the semantic level, the system can suggest candidate **converters** in much the same way it suggests candidate producers and consumers, but for one important difference: multiple **Activities** can be put in sequence to form a **conversion chain**.

Given a mismatch between a source type `X` and a target type `Y`, the query to look for a one-shot converter (*i.e.* only one **Activity**) is given in Listing 4.6 and illustrated on Figure 4.31. The query for a chain of two converters is given in Listing D.1 and illustrated on Figure D.1, both found in Appendix D.

Listing 4.6: `X` → `Y` Converter Search Query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cowork: <http://www.i3s.unice.fr/~cerezo/cowork/latest/cowork.rdfs#>

SELECT ?activity WHERE {
  ?activity rdf:type cowork:Activity .

  ?activity cowork:hasInput ?inputport .
  ?inputport cowork:hasSpecification ?inputdataset .
  {
    ?inputdataset rdf:type X .
  }
  UNION
  {
    ?inputdataset rdf:type ?inputtype .
    X rdfs:subClassOf ?inputtype .
  }

  ?activity cowork:hasOutput ?outputport .
  ?outputport cowork:hasSpecification ?outputdataset .
  {
    ?outputdataset rdf:type Y .
  }
  UNION
  {
    ?outputdataset rdf:type ?outputtype .
    Y rdfs:subClassOf ?outputtype .
  }
}
```

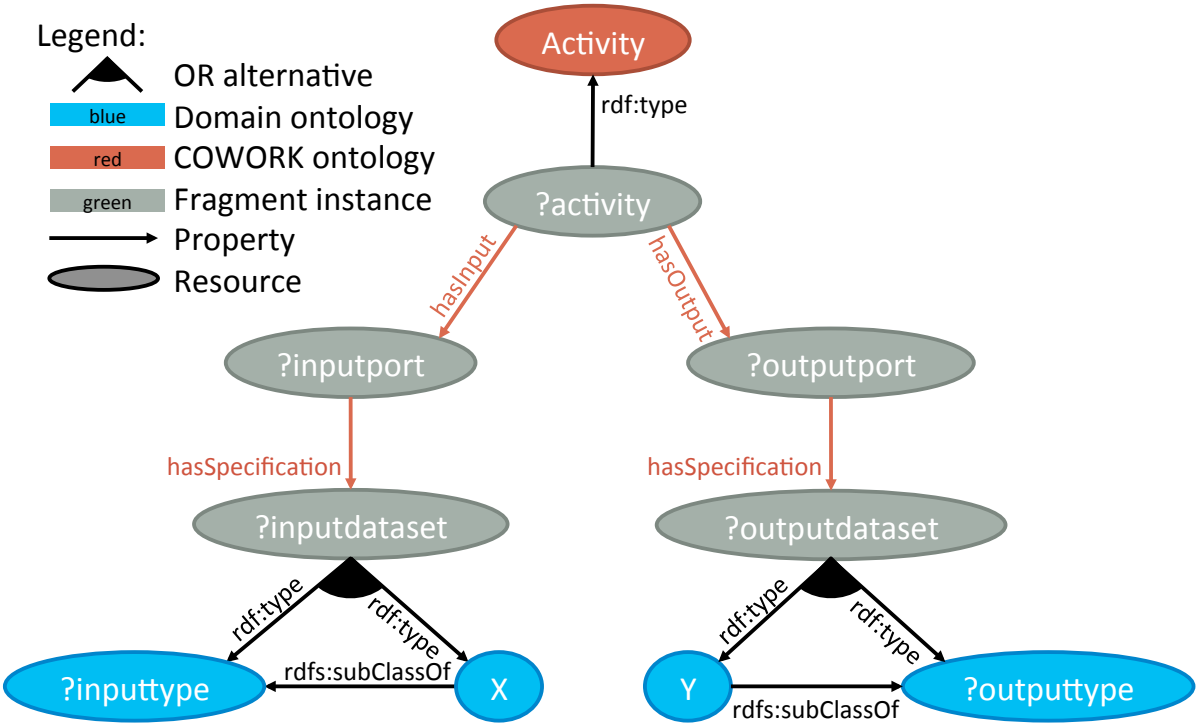


Figure 4.31: X → Y Converter Search Query

Such queries can be generated automatically up to any number of intermedirary steps. The optimal number of conversion steps is unknown, but each intermediary type gone through increases the likelihood of data loss or alteration. Therefore, the system looks for converter chains up to an arbitrary length threshold. That threshold should be given a default value and should be modifiable by the users if they so wish.

Once candidate converters (and conversion chains of size up to the threshold) have been retrieved, they are ranked according to the following principles:

- **penalize partial matches**: the more **Specifications** a converter tackles at once, the higher it should rank;
- **penalize chain length**: the longer the chain, the more likely data will be lost or altered, hence the need for longer chains to rank lower; and
- **penalize extra functions and concerns**: any **Function** or **Concern** that is not identified by the system as relating to conversion is a sign that the converter (or chain of converters) might perform unwanted operations on the data, maybe causing data loss or even altering data semantics, hence the fewer such **Annotations** there are, the higher the rank.

The *convert* function detailed in Algorithm 9 finds candidate converters (and conversion chains) for a selected mismatched **Data Link** and ranks them based on the aforementioned principles.

Algorithm 9 Suggesting Converters

function CONVERT($dl : DataLink$)
  $result : Dictionary(List(Activity), Float)$ ▷ Initialize result list.
  for all $d_s \in \Gamma_S(source(dl))$ **do** ▷ $\forall d_s$ Specification of the OutputPort source of dl:
    for all $d_t \in \Gamma_S(target(dl))$ **do** ▷ $\forall d_t$ Specification of the InputPort target of dl:
      for all $c \in converters(d_s, d_t)$ **do** ▷ $\forall c$ converter (or chain) from d_s to d_t:
        $w : Float$ ▷ Initialize weight.
        $w \leftarrow max_{d_i \in \Gamma_S(inputs(c[first]))} score(d_i, d_s)$ ▷ Compute best score of inputs
        $w \leftarrow w * max_{d_o \in \Gamma_S(outputs(c[last]))} score(d_t, d_o)$ ▷ and adjust with outputs'.
        if $c \in keys(result)$ **then** ▷ If c is already in the result list,
          $result[c] \leftarrow result[c] + w$ ▷ then update its weight;
        else
          $result[c] \leftarrow w$ ▷ else add c to the list with weight w.
        end if
      end for
    end for
  end for
  for $(c, w) \in result$ **do** ▷ $\forall (a, w)$ converter (or chain) and associated weight:
    $result[c] \leftarrow \frac{result[c]}{|\Gamma_S(source(dl))| * |\Gamma_S(target(dl))|}$ ▷ Penalize partial matches.
    $result[c] \leftarrow \frac{result[c]}{size(c)}$ ▷ Penalize chain length.
    $result[c] \leftarrow \frac{result[c]}{nonConvert(c)}$ ▷ Penalize extra Functions and Concerns.
  end for
  return $sort(result)$ ▷ Sort by decreasing weight.
end function

It uses the $score$ function defined in Section 4.4.3.2, both with the standard order or parameters, when it scores ouput **Datasets**, and in reverse, when it scores input **Datasets**, for the same reasons consumers do. It also uses the previously defined $inputs(a : Activity)$ and $outputs(a : Activity)$ as well as the following sub-functions:

- $converters(d_s : Dataset, d_t : Dataset)$ uses the queries given in Listing 4.6 and Listing D.1 as well as longer queries auto-generated up to the threshold, to retrieve candidate converters and conversion chains from d_s to d_t;
- $size(l : List)$ returns the size of l; and
- $nonConvert(c : List(Activity))$ returns the list of **Functions** and **Concerns** contained in the conversion chain that are not identified as pertaining to conversion.

4.6 Conversion

After transforming a high-level **Conceptual Workflow** into an **Intermediate Representation** through **Mapping**, the **Conversion** step transforms that representation into one that a third-party scientific workflow framework will be able to handle and enact: that translation step is performed completely automatically by the system.

The reason why that step can be automated, when the previous one (*i.e.* **Mapping**) can not, is that an **Intermediate Representation**, though part of the **Conceptual Workflow Model**, contains a full-fledged abstract workflow. **Conversion** is the process of extracting that abstract workflow and translating it into a target scientific workflow language.

As of this writing, our system supports three target languages:

- Section 4.6.2 details the **Conversion** to GWENDIA [Montagnat 09], the language of MOTEUR [Glatard 08] (*cf.* Section A.7), chosen because it is the language used on the VIP platform we relied on for use cases (*cf.* Section 5.2);
- Section 4.6.3 details the **Conversion** to `t2flow`, the language of the `version 2` of Taverna [Missier 10a] (*cf.* Section A.11), chosen because Taverna is a very well-known scientific workflow framework with a big user community; and
- Section 4.6.4 details the **Conversion** to IWIR [Plankensteiner 11], the fine-grained interoperability language of the SHIWA [Krefting 11] platform (*cf.* Section A.9), chosen because it is meant to enable enactment on many different scientific workflow frameworks.

Since all three currently supported targets, as well as many other scientific workflow languages, comply with the eXtensible Markup Language (XML), Section 4.6.1 very briefly introduces that meta-format.

4.6.1 XML in a nutshell

XML is a World Wide Web Consortium (W3C) standard which defines not one file format, but a meta-format: a set of rules for encoding documents in a way that is easy for machines to process yet readable for humans (though extremely verbose).

Complying with the rules of XML when creating a file format, no matter its target contents, presents the advantage of relying on now quite robust technologies to parse, produce, transmit, transform and validate the resulting XML files.

An XML file is essentially a markup tree structure:

- nodes start with a beginning markup `<node-name>` and end with `</node-name>`;
- leaf nodes can be abbreviated into a self-closing markup `<node-name/>`;
- children of a node are declared between its delimiting markups; and
- nodes bear attributes that are declared in the beginning markup like so: `<node-name attribute1="value1" attribute2="value2" ...>`.

We only described here the bare minimum needed to follow the **Conversion** processes described thereafter. The full specification[1] is hosted by the W3C and the reader willing to dive into XML might find the w3schools tutorial[2] especially useful.

4.6.2 To GWENDIA (MOTEUR)

A GWENDIA [Montagnat 09] file is an XML document with a root `<workflow>` node which has a `name` attribute and four children nodes, as summed up on Figure 4.32:

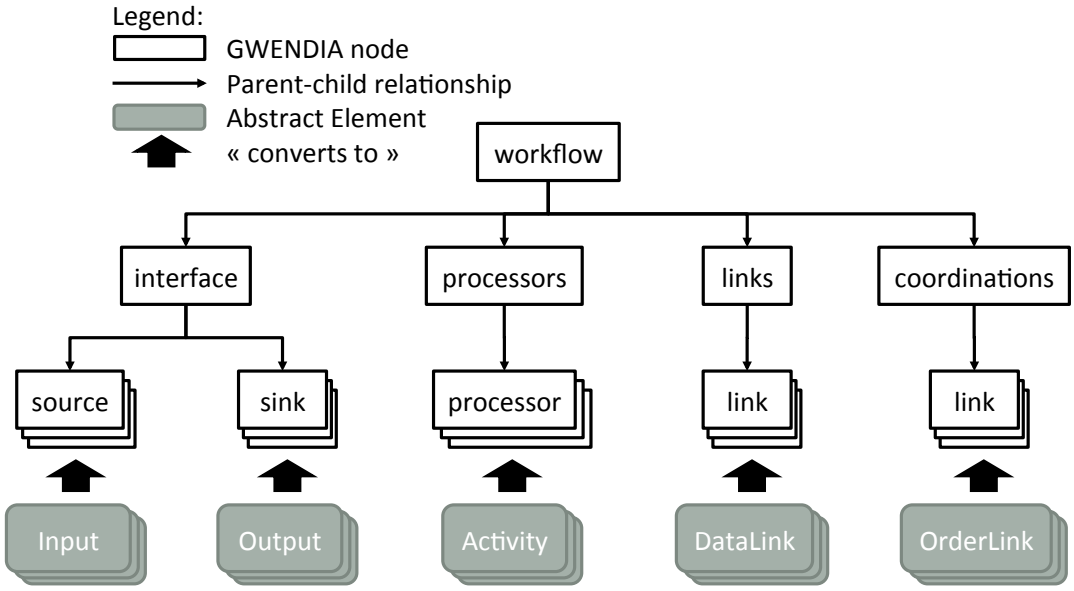


Figure 4.32: Conversion to GWENDIA - Basic Structure

- `<interface>` is the parent node of any number of `<source>` nodes which represent data entry points and correspond to **Inputs**; and `<sink>` nodes which represent data exit points and correspond to **Outputs**;
- `<processors>` is the parent node of any number of `<processor>` nodes, which themselves contain `<in>` and `<out>` nodes for **Ports** and an `iterationstrategy` node, each of which corresponds to an **Activity**;
- `<links>` is the parent node of any number of `<link>` nodes, each of which corresponds to a **Data Link**; and
- `<coordinations>` is the parent node of any number of `<link>` nodes, each of which corresponds to an **Order Link**.

[1] XML Specification: `http://www.w3schools.com/xml/`
[2] w3schools.com XML tutorial: `http://www.w3schools.com/xml/`

4.6.2.1 Converting Inputs/Outputs

Inputs are converted to `<source>` nodes and **Outputs** and converted to `<sink>` nodes, but there is not necessarily a one-to-one relationship between both sides. Indeed, `<source>` and `<sink>` nodes each represent one piece (or collection) of data at a time. Therefore, an **Input** (resp. **Output**) with two or more **Output Ports** (resp. **Input Ports**) will be converted into that many `<source>` (resp. `<sink>`) nodes.

For instance, if the **Intermediate Representation** features three **Inputs** `A`, `B` and `C` with one **Output Port** each and one **Output** `D` with two **Input Ports**, as shown on Figure 4.33a, that will translate to three `<source>` and two `<sink>` nodes, as shown on Figure 4.33b and detailed on Listing 4.7.



(a) Intermediate Representation (b) MOTEUR screenshot

Figure 4.33: Conversion to GWENDIA - Inputs/Outputs Example

Listing 4.7: Conversion to GWENDIA - Inputs/Outputs Example

```
<workflow name="inputs/outputs_example">
  <interface>
    <source name="A" type="string"/>
    <source name="B" type="string"/>
    <source name="C" type="string"/>
    <sink name="D_in0" type="string"/>
    <sink name="D_in1" type="string"/>
  </interface>
  <processors></processors><links></links><coordinations></coordinations>
</workflow>
```

By convention, when an **Input** or **Output** has only one **Port**, the resulting node inherits its name. When there are two or more **Ports**, each resulting node is named with the concatenation of the **Input** (or **Output**) name, an underscore and the corresponding **Port** name, so as to differentiate them.

4.6.2.2 Converting Activities

Activities with both **Input Ports** and **Output Ports**, *i.e.* **Activities** other than **Inputs** and **Outputs** are converted to `<processor>` nodes. Each **Input Port** becomes an `<in>` child node and each **Output Port** an `<out>` child node.



(a) Intermediate Representation (b) MOTEUR screenshot

Figure 4.34: Conversion to GWENDIA - Activities Example

For instance, if the **Intermediate Representation** features three **Activities** `P`, `Q` and `R` with one **Input Port** and one **Output Port** each and an **Activity** `S` with two **Input Ports** and two **Output Ports**, as shown on Figure 4.34a, it will translate to four `<processor>` nodes as shown on Figure 4.34b and detailed on Listing 4.8.

When an **Activity** has two or more **Input Ports**, the associated **Iteration Strategy** is converted into an `<iterationstrategy>` child node, with its tree of operators translated into a tree of corresponding nodes. For instance, $a \otimes (b \odot c)$ would translate to:

```
<cross>
  <port name="a" />
  <dot>
  <port name="b" />
<port name="c" />
</dot>
</cross>
```

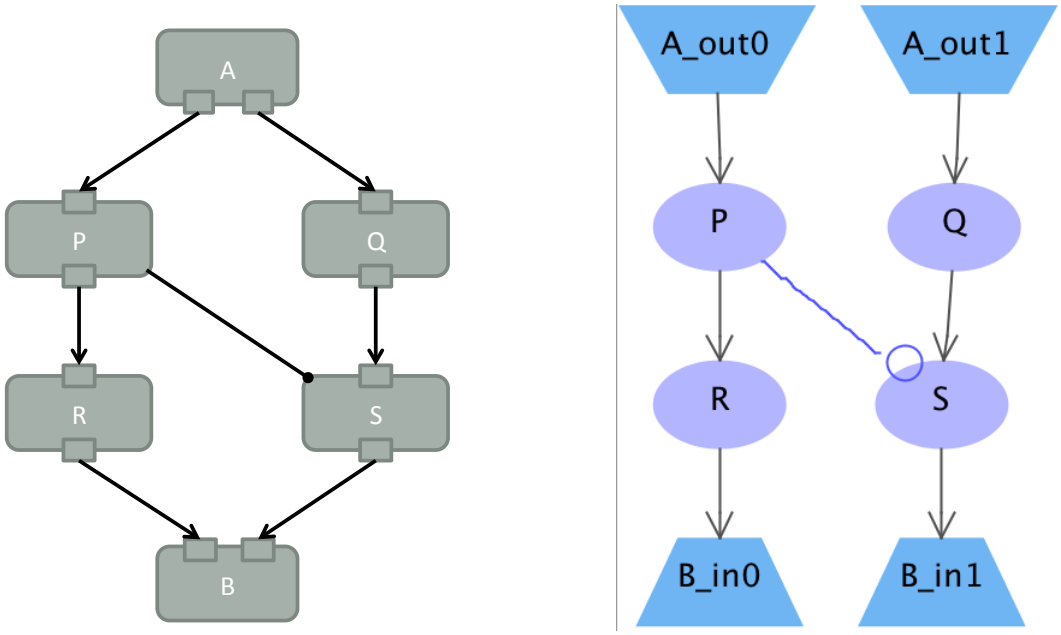
Listing 4.8: Conversion to GWENDIA - Activities Example

```
<workflow name="activities_example">
  <interface></interface>
  <processors>
    <processor name="P">
      <in depth="0" name="in" type="string"/>
      <out depth="0" name="out" type="string"/>
    </processor>
    <processor name="Q">
      <in depth="0" name="in" type="string"/>
      <out depth="0" name="out" type="string"/>
    </processor>
    <processor name="R">
      <in depth="0" name="in" type="string"/>
      <out depth="0" name="out" type="string"/>
    </processor>
    <processor name="S">
      <in depth="0" name="in0" type="string"/>
      <in depth="0" name="in1" type="string"/>
      <out depth="0" name="out0" type="string"/>
      <out depth="0" name="out1" type="string"/>
      <iterationstrategy>
        <cross>
          <port name="in0"/>
          <port name="in1"/>
        </cross>
      </iterationstrategy>
    </processor>
  </processors>
  <links></links><coordinations></coordinations>
</workflow>
```

4.6.2.3 Converting Links

Data Links and **Order Links** are converted to `<link>` nodes, but they are not placed inside the same parent nodes: **Data Links** become children nodes of `<link>`, whereas **Order Links** become children nodes of `<coordination>`. Each `<link>` has a `from` attribute which references the name of the link source and a `to` attribute for the target.

For instance, if an **Intermediate Representation** features, as shown on Figure 4.35a: one **Input** `A`; four **Activities** `P`, `Q`, `R` and `S`; an **Output** `B`; **Data Links** `A` to `P` and `Q`, from `P` to `R`, from `Q` to `S` and from `R` and `S` to `B`; as well as an **Order Link** from `P` to `S`; then those links will translate to six children nodes for `<links>` and one child node for `<coordination>`, as shown on Figure 4.35b and detailed on Listing 4.9.



(a) Intermediate Representation (b) MOTEUR screenshot

Figure 4.35: Conversion to GWENDIA - Links Example

Listing 4.9: Conversion to GWENDIA - Links Example

```
<workflow name="links_example">
  <interface>
    <source name="A_out0" type="string"/>
    <source name="A_out1" type="string"/>
    <sink name="B_in0" type="string"/>
    <sink name="B_in1" type="string"/>
  </interface>
  <processors>
    <processor name="P">
      <in depth="0" name="in" type="string"/>
      <out depth="0" name="out" type="string"/>
    </processor>
    <processor name="Q">
      <in depth="0" name="in" type="string"/>
      <out depth="0" name="out" type="string"/>
    </processor>
    <processor name="S">
      <in depth="0" name="in" type="string"/>
      <out depth="0" name="out" type="string"/>
    </processor>
    <processor name="R">
      <in depth="0" name="in" type="string"/>
      <out depth="0" name="out" type="string"/>
    </processor>
  </processors>
  <links>
    <link from="A_out0" to="P:in"/>
    <link from="A_out1" to="Q:in"/>
    <link from="Q:out" to="S:in"/>
    <link from="P:out" to="R:in"/>
    <link from="R:out" to="B_in0"/>
    <link from="S:out" to="B_in1"/>
  </links>
  <coordinations>
    <link from="P" to="S"/>
  </coordinations>
</workflow>
```

4.6.3 To t2flow (Taverna 2)

The first version of Taverna [Missier 10a] uses a language called SCUFL [Oinn 04] and the team behind its development announced that the future version 3 will use an evolution of it called "*SCUFL2*"[3]. `t2flow` is merely a serialization format for `Taverna 2` workflows.

Like GWENDIA, `t2flow` is XML-based. However, it is considerably more verbose, since it is a serialization format, *i.e.* a direct reproduction, without any simplification, of the complete structure of objects involved in instances of `Taverna 2` workflows. `t2flow` documents have a root node called `<workflow>`, but also an additional layer of `dataflow` nodes. The reason for that is the way Taverna handles encapsulation: each sub-workflows is reproduced in its entirety in its own `<dataflow>` node.

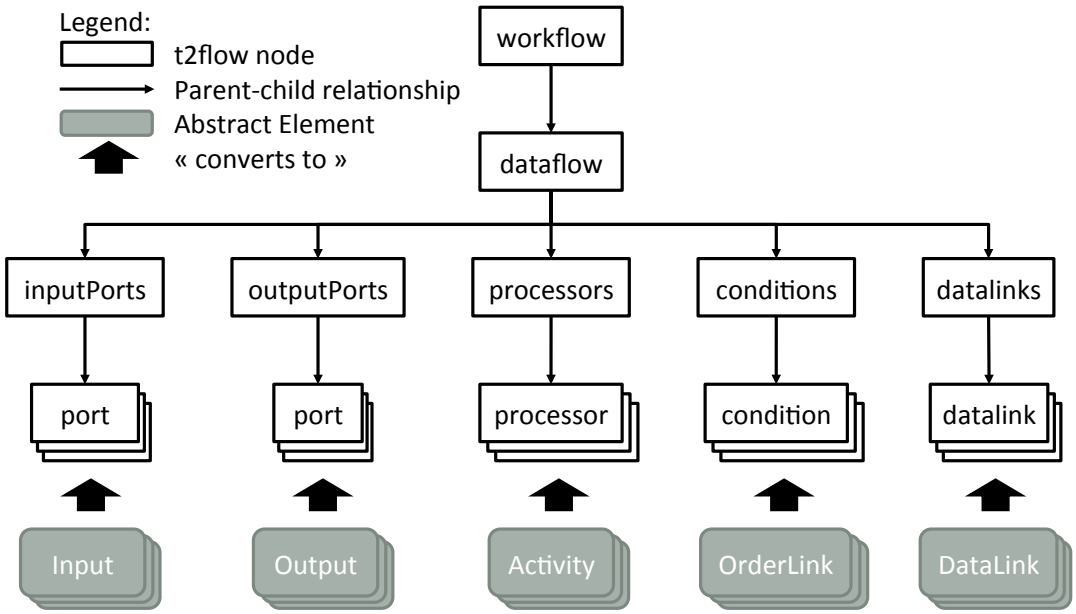


Figure 4.36: Conversion to t2flow - Basic Structure

`<dataflow>` nodes have six children, as summed up on Figure 4.36:

- `<name>` contains the name of the workflow;
- `<inputPorts>` is the parent node of any number of `<port>` nodes which represent data entry points and correspond to **Inputs**;
- `<outputPorts>` is the parent node of any number of `<port>` nodes which represent data exit points and correspond to **Outputs**;
- `<processors>` is the parent node of any number of `<processor>` nodes, each of which corresponds to an **Activity**;
- `<conditions>` is the parent node of any number of `<condition>` nodes, each of which corresponds to an **Order Link**; and
- `<datalinks>` is the parent node of any number of `<datalink>` nodes, each of which corresponds to a **Data Link**.

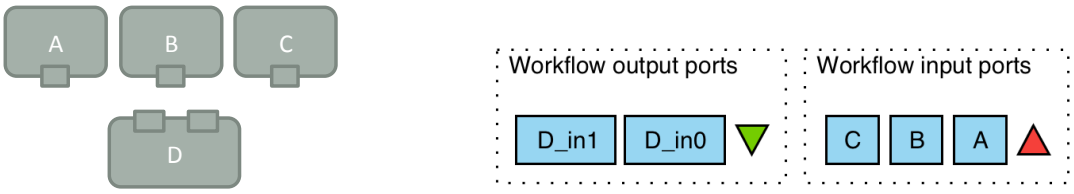
[3]SCUFL2 Announcement: `http://bit.ly/scufl2announcement`

Because the structure of `t2flow` is close to that of GWENDIA (*cf.* Section 4.6.2), the two **Conversion** processes are similar as well. The main differences pertain to arbitrary choices (*e.g.* GWENDIA uses attributes where `t2flow` uses text content) and the nature of `t2flow`: because it is a serialization format, the actual structure features a lot of repetition (*i.e.* the same data reproduced in multiple places) as well as many additional layers.

4.6.3.1 Converting Inputs/Outputs

Inputs (resp. **Outputs**) are converted to `<port>` nodes, children of `<inputPorts>` (resp. `<outputPorts>`). As with GWENDIA, **Inputs** (resp. **Outputs**) with two or more **Output Ports** (resp. **Input Ports**) are converted into as many `<port>` nodes.

Again with the example of an **Intermediate Representation** featuring three **Inputs** `A`, `B` and `C` with one **Output Port** each and one **Output** `D` with two **Input Ports**, as shown on Figure 4.37a, **Conversion** will produce three `<port>` nodes inside `<inputPorts>` and two inside `<outputPort>`, as shown on Figure 4.37b and detailed on Listing E.1 (found in Appendix E).



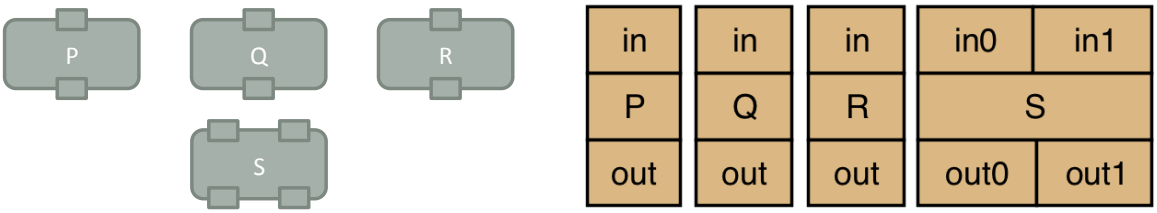
(a) Intermediate Representation (b) Taverna 2 screenshot

Figure 4.37: Conversion to t2flow - Inputs/Outputs Example

4.6.3.2 Converting Activities

Activities with both **Input Ports** and **Output Ports**, *i.e.* **Activities** other than **Inputs** and **Outputs**, are converted to `<processor>` nodes. Each **Input Port** (resp. **Output Port** becomes a `<port>` inside the `<inputPorts>` (resp. `<outputPorts>` child node, but they are also described deeper in the tree and both descriptions are mapped to one another by `<map>` nodes children of `<inputMap>` (resp. `<outputMap>`).

Back to the example of and **Intermediate Representation** featuring three **Activities** `P`, `Q` and `R` with one **Input Port** and one **Output Port** each and an **Activity** `S` with two of each, as shown on Figure 4.38a, **Conversion** will produce four `<processor>` nodes as shown on Figure 4.38b and detailed on Listing E.2 (found in Appendix E).



(a) Intermediate Representation (b) Taverna 2 screenshot

Figure 4.38: Conversion to t2flow - Activities Example

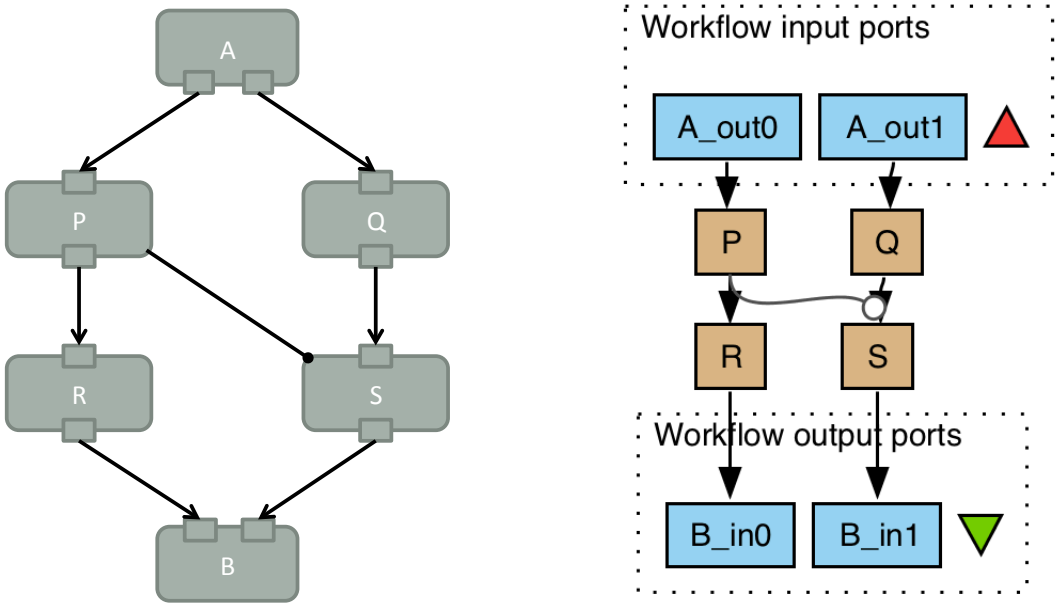
When an **Activity** has two or more **Input Ports**, the operators of the **Iteration Strategy** are converted into a tree of corresponding nodes in the `<iterationStrategyStack>` `<iteration><strategy>` branch of nodes contained in the `<processor>` node. For instance, $a \otimes (b \odot c)$ would translate to:

```
<strategy>
  <cross>
    <port name="a" depth="0" />
    <dot>
      <port name="b" depth="0" />
      <port name="c" depth="0" />
    </dot>
  </cross>
</strategy>
```

4.6.3.3 Converting Links

Order Links (resp. **Data Links**) are converted to `<condition>` (resp. `<datalink>`) children nodes of `<conditions>` (resp. `datalinks`). `<condition>` nodes have two attributes: `control` is the source of the **Order Link** and `target` its target.

Each `<datalink>` node has two children nodes `<sink>` and `<source>` describing the target and source respectively. If the target (resp. source) is a **Port** belonging to an **Output** (resp. Input), then it is described through a single `<port>` node, otherwise it is converted to a `<processor>` specifying which **Activity** the port belongs to and a `<port>` node specifying which **Port** is the target (resp. source).



(a) Intermediate Representation (b) Taverna 2 screenshot

Figure 4.39: Conversion to t2flow - Links Example

Back to the example of **Intermediate Representation** described in Section 4.6.2.3, **Conversion** will produce one `<coordination>` node and six `<datalink>` nodes, as shown on Figure 4.39b and detailed on Listing E.3, (found in Appendix E).

4.6.4 To IWIR (SHIWA)

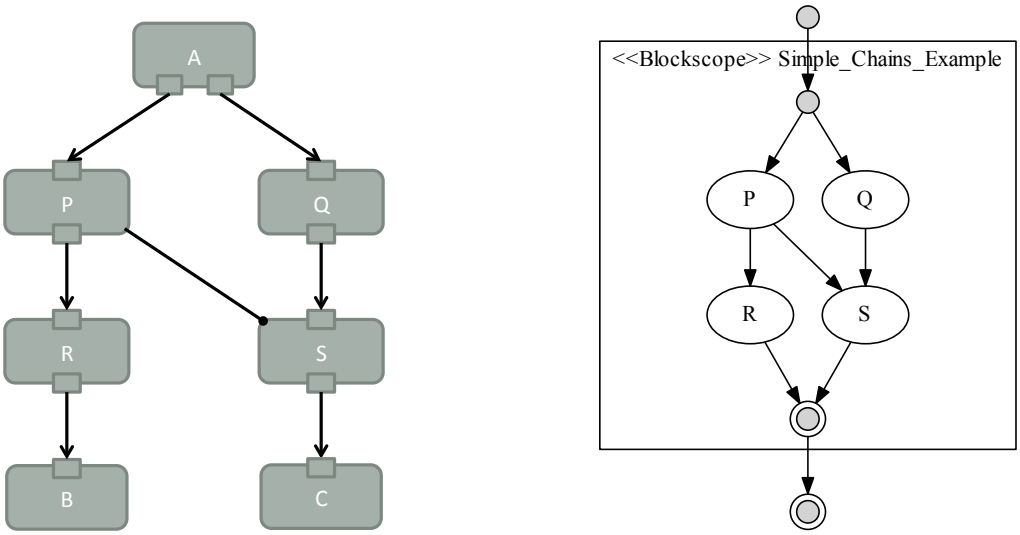
The SHIWA [Krefting 11] platform implements scientific workflow interoperability either at a coarse-grained level, with each "*non-native*" sub-workflow packaged with the corresponding "*non-native*" enactor into a "*job*" of a "*meta-workflow*", or at a fine-grained level through conversion to a common pivot language: IWIR [Plankensteiner 11].

Each scientific workflow framework participating in the project provides both conversion from their scientific workflow model to the pivot language and direct interpretation of it by their enactor. As a result, barring technical incompatibilities, workflows can be run by "*non-native*" enactors. The full specification of IWIR was made publicly available[4] as well as a dedicated library[5] to help developers add IWIR-support to their scientific workflow framework.

The **Conversion** from an **Intermediate Representation** to IWIR is more than a question of format. Indeed, as explained in Chapter 3, the **Conceptual Workflow Model** is a hybrid model, whereas IWIR is a control-driven model.

4.6.4.1 Converting Simple Chains

When working with simple chains, *i.e.* when each **Activity** has only one **Input**, the differences between IWIR and data-driven languages (like those converted to in Section 4.6.2 and Section 4.6.3) appear deceptively few.



(a) Intermediate Representation

(b) Graph generated by IWIRtool

Figure 4.40: Conversion to IWIR - Simple Chains Example

The most notable difference is the apparition of "*scopes*": "*tasks*", "*inputs*", "*outputs*" and "*links*" all belonging to a specific parent scope and though links can target the ports of a direct sub-task, they cannot go further down the tree, *e.g.* targeting the port of a sub-task of a sub-task. With simple chains, however, there is only one scope: a `BlockScope` containing the entire workflow. Another difference is that IWIR does not differentiate much between **Data Links** and **Order Links**: the former simply link "*ports*" together, whereas the latter link "*tasks*".

[4]IWIR Specification v1.1: `http://bit.ly/IWIRspec`
[5]IWIRtool Library: `http://sourceforge.net/projects/iwirtool/`

To illustrate those differences, let us get back to a slightly modified version of the example of **Intermediate Representation** described in Section 4.6.2.3 and used in Section 4.6.3.3 as well. The only variation being that the final **Output** is split in two, as shown on Figure 4.40a.

Listing 4.10: Conversion to IWIR - Simple Chains Example

```
<IWIR version="1.1" wfname="Simple_Chains_Example" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="Simple_Chains_Example">
    <inputPorts>
      <inputPort name="A_out1" type="string"/>
      <inputPort name="A_out2" type="string"/>
    </inputPorts>
    <body>
      <task name="S" tasktype="S">
        <inputPorts>
          <inputPort name="in" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out" type="string"/>
        </outputPorts>
      </task>
      <task name="P" tasktype="P">
        <inputPorts>
          <inputPort name="in" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out" type="string"/>
        </outputPorts>
      </task>
      <task name="Q" tasktype="Q">
        <inputPorts>
          <inputPort name="in" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out" type="string"/>
        </outputPorts>
      </task>
      <task name="R" tasktype="R">
        <inputPorts>
          <inputPort name="in" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out" type="string"/>
        </outputPorts>
      </task>
    </body>
    <outputPorts>
      <outputPort name="B_in1" type="string"/>
      <outputPort name="B_in2" type="string"/>
    </outputPorts>
    <links>
      <link from="Simple_Chains_Example/A_out1" to="P/in"/>
      <link from="Simple_Chains_Example/A_out2" to="Q/in"/>
      <link from="P/out" to="R/in"/>
      <link from="Q/out" to="S/in"/>
      <link from="P" to="S"/>
      <link from="R/out" to="Simple_Chains_Example/B_in1"/>
      <link from="S/out" to="Simple_Chains_Example/B_in2"/>
    </links>
  </blockScope>
</IWIR>
```

Conversion will produce 4 tasks and 7 links, as shown on Figure 4.40b and detailed on Listing 4.10. The reason for splitting the **Output** is to emphasize the differences between the graphical representations: the links on the IWIR representation are all control links and the pairs of circles at the top and bottom of the graph represent `start` and `stop` states, not inputs/outputs.

4.6.4.2 Iteration strategies

As explained in Section 3.2.4, **Iteration Strategies** are necessary to handle "*implicit loops*": when an **Activity** with two or more **Input Ports** is provided with input data of a higher depth than it is meant to process, the enactor can iterate automatically over the data, based on the **Iteration Strategy**. However, that behavior is typically driven by data and would make no sense with a control-driven model.

Indeed, in IWIR, not only must loops be explicitly declared as "*scopes*", there are a variety of them to choose from which behave differently: the traditional `for`, `forEach` and `while` as well as the parallel computing variants `parallelFor` and `parallelForEach`.

Therefore, **Iteration Strategies** must be converted into explicit loops and the best match is `parallelForEach`, which will iterate over one of the **Input Ports** data and process different data items simultaneously, if possible. The **Conversion** will depend not only on the **Iteration Strategy**, but on the depths of the concerned **Input Ports**: where a list will require a single loop, a list of lists will require two loops and so on and so forth.

The dot operator is converted into as many loops as there are extra depth levels on each of the input data. Note: they must all have the same number of extra depth levels for the dot operator to work. For instance, with a, b and c each provided with data 2 depths higher than expected, the **Iteration Strategy** $a \odot b \odot c$ will be converted to 2 `parallelForEach` loops, each decrementing all three port data depths by one simultaneously.

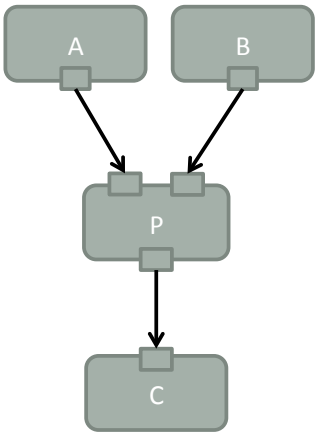
The cross operator is converted into as many loops as the sum of extra depth levels on each of the input data. For instance, with a and b provided with data 1 depth higher than expected and c 2 depths higher, $a \otimes b \otimes c$ will be converted to 4 `parallelForEach` loops, each decrementing one data depth by one.

Example Let us illustrate the **Conversion** of **Iteration Strategies** with a minimal example, shown on Figure 4.41a: two **Inputs** `A` and `B`, one **Activity** `P` and one **Output** `C`. Since depths can be increased virtually indefinitely, there is an infinity of cases. Let us focus on the following three, with all three corresponding listings found in Appendix F:

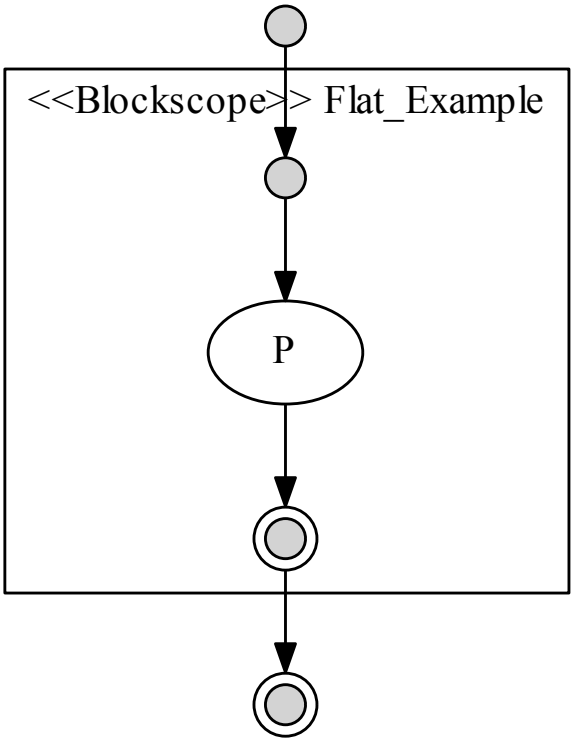
Flat Inputs Example
If both `A` and `B` match the depths of the **Input Port** they are bound to, then the **Iteration Strategy** is ignored, as shown on Figure 4.41b and detailed on Listing F.1.

Dot Example
If both `A` and `B` are one depth level above that of the **Input Port** they are bound to and the **Iteration Strategy** is $A \odot B$, then the **Conversion** generates 1 `parallelForEach` loop, as shown on Figure 4.41c and detailed on Listing F.2.

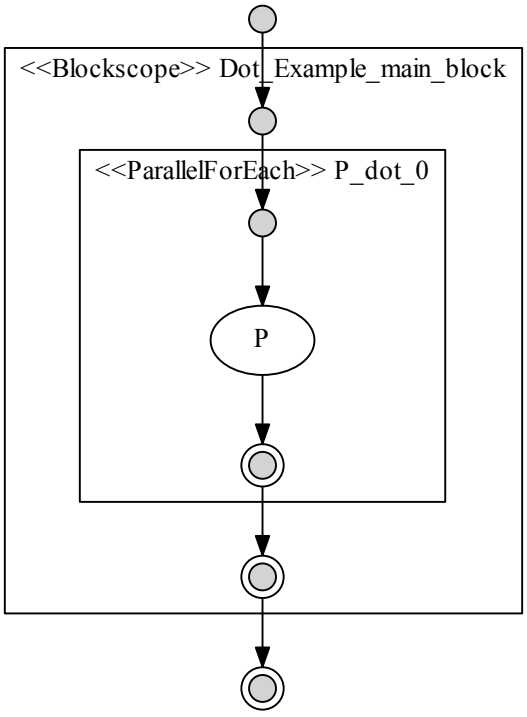
Cross Example
With the same depths as the previous **Dot Example**, but this time with an **Iteration Strategy** of $A \otimes B$, the **Conversion** generates 2 `parallelForEach` loops, as shown on Figure 4.41d and detailed on Listing F.3.



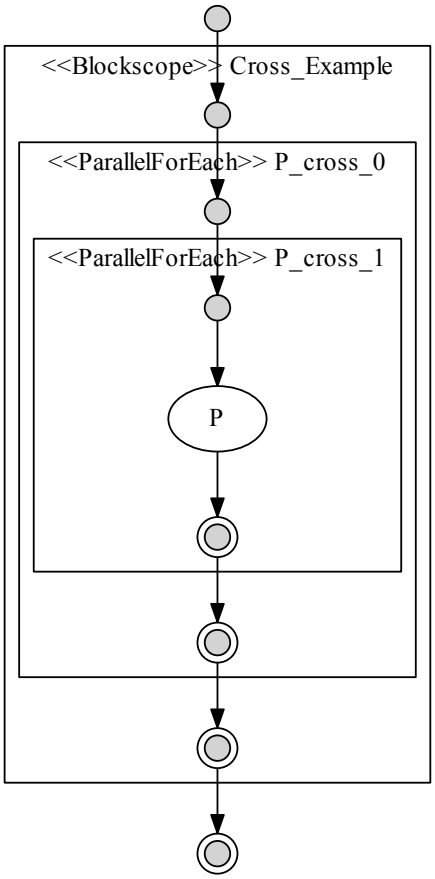
(a) Intermediate Representation



(b) Flat Inputs - Graph generated by IWIRtool



(c) Dot Strategy - Graph generated by IWIRtool



(d) Cross Strategy - Graph generated by IWIRtool

Figure 4.41: Converting Iteration Strategies to IWIR

4.6.5 Classification

The position of **Conversion** in the model transformation taxonomy defined in [Czarnecki 03] and described in Section 2.3.3, is represented on Figure 4.42. Like the classification of **Mapping**, it is a Feature Model instance and every node on the graph is relevant to **Conversion**:

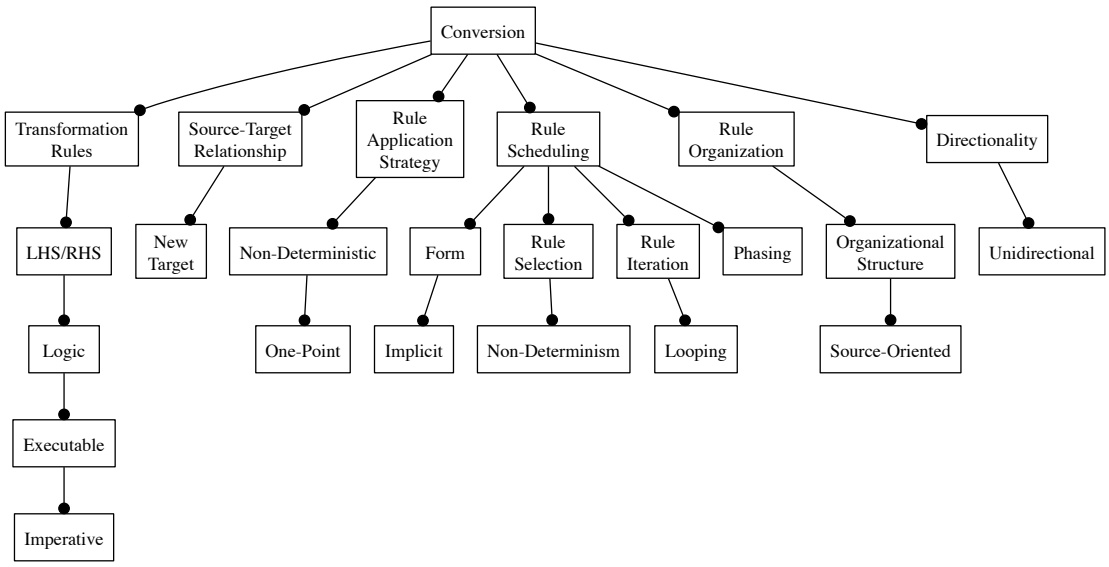


Figure 4.42: Classification of the Conversion model transformation

- **Transformation Rules**: the **Conversion** algorithms are themselves written in Java (*i.e.* the language adopted for the prototype), but in a mostly imperative fashion;

- **Source-Target Relationship**: a new target abstract workflow is built based on the source **Intermediate Representation**;

- **Rule Scheduling and Application Strategy**: the **Intermediate Representation** to convert is treated sequentially in RDF, iterating over all triples, but triples themselves are not ordered, so the rules (based on the nature of the triple in question) are applied in non-deterministic order, one at a time, though there are some limitations (some triples must be treated before others) inducing phases in the tranformation;

- **Rule Organization**: the rules are source-based in that each triple will be translated in a given way, based on its subject, predicate and object; and

- **Directionality**: **Conversion** transforms from **Intermediate Representations** to abstract workflows and the reverse transformation is essentially impossible, since high-level information cannot be inferred automatically (hence the need for semantic representations).

4.6.6 Discussion

Conversion is limited both at the level of the model and at that of the underlying infrastructure.

Model Limitations

The closer the target scientific workflow model is to the **Conceptual Workflow Model**, the simpler **Conversion** becomes. For models of a different nature, *e.g.* control-driven models or petri nets, **Conversion** means a lot more than translating between file formats.

There are two distinct model-level issues with **Conversion**: **expressivity** issues and **missing information** issues.

Expressivity issues arise when the target scientific workflow model lacks features that are used in the converted **Conceptual Workflow**. Theoritically, it should always be possible to inject the missing features into the target language through *ad-hoc* dedicated **Activities**. However, the case can be made that it would require a lot of work to build and maintain all those **Activities** and that they would decrease the legibility and flexibility of the resulting workflow.

It is important to note that a model may be more expressive than the **Conceptual Workflow Model** in general, yet lack some of its features. For instance, IWIR [Plankensteiner 11] is largely more expressive than the **Conceptual Workflow Model** but lacks **Iteration Strategies** because they do not fit a control-driven model.

Missing information issues are, in the sense, the inverse of expressivity issues. They arise when the target language requires more information than is provided in the **Conceptual Workflow Model**. It is typically the case with petri nets where each piece of intermediate data is described by a node of its own. While missing data can probably be generated well enough to meet the syntaxic requirements of the target framework, it will likely not be as meaningful as what the users would have devised, had they created the workflow directly in the target language.

Infrastructure Limitations

Conversion issues do not stop with the scientific workflow itself. Most low-level **Activities** are meant to be run in a very specific context, *e.g.* a Linux 64bits machine or a Globus[6] grid, and will fail if run in a different context.

Most scientific workflow frameworks have been tailored, over time, to specific Distributed Computing Infrastructures (DCIs) and that is reflected in the way their **Activities** are created, described and enacted.

Though it is impractical and certainly hard to maintain, it does not seem impossible to support the idiosyncratic **Activity** management systems of each target framework. However, converting an **Activity** from one framework to another might be impossible to do at all, much less automate.

As a result, while a knowledge base could be shared by users of different scientific workflow frameworks, it is extremely unlikely that all the **Activities** in the knowledge base would be compatible with all target languages.

[6]Globus Toolkit: `http://www.globus.org/toolkit/`

CHAPTER SUMMARY

The second (out of two) **contribution** of this work is a semi-automated **Transformation Process** designed to help users transform purely conceptual workflows into executable abstract workflows with algorithms and tools: **Discovery** to look for potentially relevant Fragments in the Knowledge Base, **Weaving** to automatically apply a Fragment to a workflow, **Composition suggestion algorithms** (for links, producers, consumers and converters) to assist the composition of abstract elements and **Conversion** to automatically convert the workflow for an existing framework.

CHAPTER 5

VALIDATION

There are many ways to approach the validation of the current work, all of them ambitious in some way. We chose to focus validation on what seemed the most fundamental aspects of our contributions, *i.e.* the **Conceptual Workflow Model** itself and the building blocks of the **Transformation Process**.

In order to do so, we have built a prototype framework, described in Section 5.1 and integrated it into the Virtual Imaging Platform (VIP), *cf.* Section 5.2. Section 5.3 details our validation of the **Conceptual Workflow Model** and Section 5.4 that of the **Transformation Process**, based on use cases taken from the VIP project.

5.1 Prototype

To ensure that the **Conceptual Workflow Model** and that the algorithms used in the **Transformation Process** are sound, we built a prototype framework, `cowork`, to test them in isolation as well as against real-life examples, taken from the VIP project. The prototype can be freely obtained as explained on the MODALIS team wiki[1].

5.1.1 Architecture

The `cowork` prototype is coded in Java, is built using and is split into 6 modules, as illustrated on Figure 5.1:

- `gui` is the graphical interface letting users design **Conceptual Workflows** and access the features of the prototype; it relies on Java Swing, the primary Java Graphical User Interface (GUI) library;
- `model` handles the **Conceptual Workflow Model** itself; it relies on Apache Jena RDF API[2], a well-known Java semantic library;
- `annotation` is a sub-module of `model` and is dedicated to **Annotations**; it also relies on Apache Jena RDF API;
- `mapping` contains the tools and algorithms used for the **Mapping** process, it relies on Apache Jena ARQ[3], the Apache Jena SPARQL Protocol and RDF Query Language (SPARQL) engine;

[1] How to get the prototype: `https://modalis.i3s.unice.fr/softwares/cowork`
[2] Apache Jena RDF API: `http://jena.apache.org/documentation/rdf/index.html`
[3] Apache Jena ARQ: `http://jena.apache.org/documentation/query/index.html`

- `conversion` contains the **Conversion** algorithms towards each target language; it relies on basic Java XML libraries for GWENDIA (MOTEUR) and t2flow (Taverna), and on the IWIRtool library for IWIR (SHIWA); and
- `knowledgebase` interfaces with the knowledge base queried by the **Discovery** and **Composition** processes and used to store **Fragments**; it relies on Apache Jena SDB[4] to store triples in a MySQL[5] relational database.

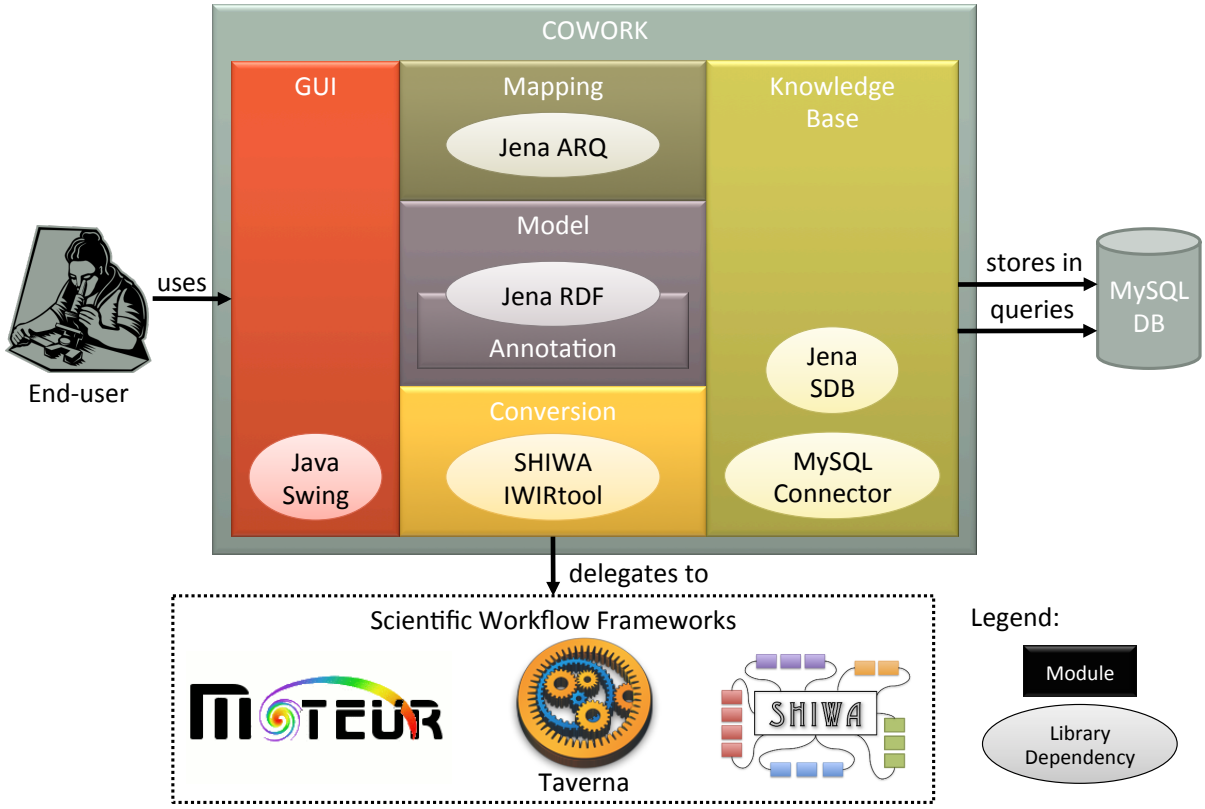


Figure 5.1: Prototype Architecture

5.1.2 Features

Here is the list of features implemented in the prototype, as of `version 0.2`:

- Creation and edition of **Conceptual Elements** (*cf.* Section 3.1)
- Creation and edition of **Abstract Elements** (*cf.* Section 3.2)
- Addition and removal of **Annotations** by URI (*cf.* Section 3.3)
- Optional edition of the **Pattern** (*cf.* Section 3.4)
- **Weaving** (*cf.* Section 4.2)
- **Erasing** and **Merging** tools (*cf.* Section 4.3)

[4]Apache Jena SDB: `http://jena.apache.org/documentation/sdb/index.html`
[5]MySQL: `http://www.mysql.com/`

- **Discovery** functions (*cf.* Section 4.4)
- **Composition** functions (*cf.* Section 4.5)
- **Conversion** to GWENDIA, t2flow and IWIR (*cf.* Section 4.6)
- Editor GUI
- Saving to and loading from files
- Support for a relational database-stored knowledge base

Here is a list of planned features that have yet to be implemented, as of `version 0.2`:

- Leveraging SPARQL property paths for converter suggestion
- Mismatch detection
- Complete access to underlying features from the GUI
- Loading of taxonomies to ease the addition of **Annotations**
- Specification of conversion-related **Functions** and **Concerns**
- Import of **Activities** from existing description formats to ease their insertion
- Edition and removal of **Fragments** in the relational database knowledge base
- Support for file-stored knowledge bases
- Handling of all sub-types of **Activities** featured in the target languages

5.2 Virtual Imaging Platform

The use cases presented in this chapter all come from VIP, which is a joint project between the following partners: CEA-Leti[6], Creatis[7], MODALIS[8], VISAGES[9] and maatG-France[10]. The platform's goal is the integration of multiple modalities and object (organ and pathology) models into a cohesive medical image simulation platform [Glatard 13].

The projects's aims were:

- **interoperability** between various simulators that were never meant to interact, by using scientific workflows and **Semantic Annotations**,
- **data model federation** to handle all organ models in a standardized fashion, through the definition of the IntermediAte Model Format (IAMF) and
- **reliability and performance** on large-scale grid infrastructures needed to cope with the amount of data processed.

The platform is meant to be flexible and easy to extend with new simulators and object models. The semantic annotations used therein are mostly taken from the OntoVIP ontology developed as part of the project.

[6]CEA-Leti: `http://www.leti.fr/`
[7]Creatis: `http://www.creatis.insa-lyon.fr/`
[8]MODALIS: `http://modalis.i3s.unice.fr/`
[9]VISAGES: `https://www.irisa.fr/visages/`
[10]maatG-France: `http://www.maatg.com/`

5.2.1 OntoVIP

> *“ OntoVIP is a multi-level ontology designed to support the sharing of resources in the field of medical image simulation [Glatard 13]. This includes both data (e.g., the models used as input of the simulation, and the images resulting from it), and image simulation software.*
> *The overall conceptualization is based on DOLCE [Masolo 03] and relies on previous ontologies (OntoNeuroLOG ontology) developed in the context of the NeuroLOG project [Temal 08, Lando 07, Kassel 10, Gibaud 11]. The final version delivered at the end of the project was OntoVIP Version 0.41 [Forestier 11].*
>
> ONTOVIP DOCUMENTATION: `HTTP://BIT.LY/ONTOVIPV1DOC`

One of the classes of OntoVIP we are most interested in is `dataset-processing`: it was inherited from OntoNeuroLOG and is the super-class of all data processing classes in the ontology, which are clearly **Functions** in most simulations.

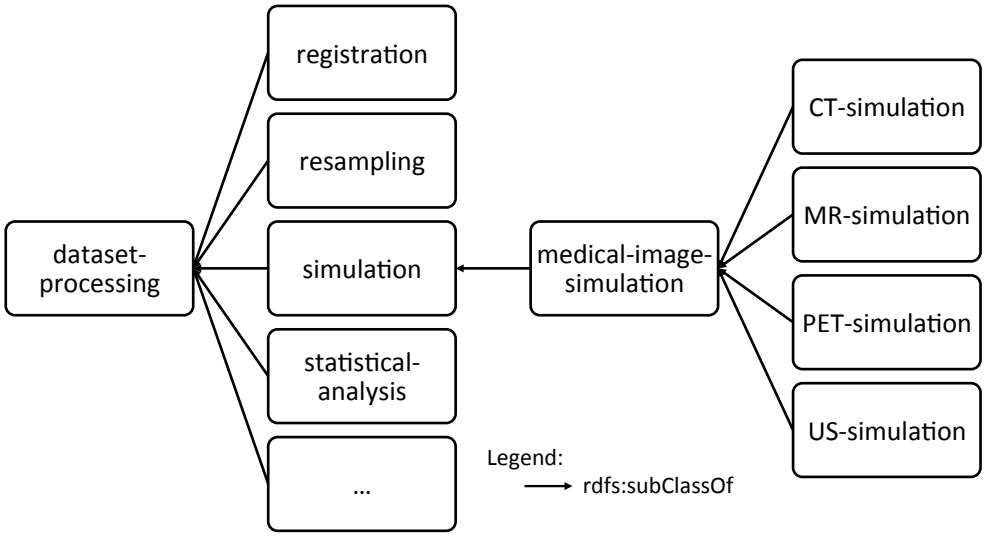


Figure 5.2: OntoVIP Excerpt - Simulations

OntoVIP `v1.0` defines 14 subclasses for `dataset-processing`, encompassing many different types of data processes, notably `registration` (which is the focus of the example given in Section 4.4.2.1) and, most importantly when it comes to the use cases we chose for our validation, `medical-imaging-simulation`, with its four subclasses, as shown on Figure 5.2, corresponding to four medical imaging modalities:

- `CT-simulation` for X-ray Computed Tomography;
- `MR-simulation` for Magnetic Resonance imaging;
- `PET-simulation` for Positron Emission Tomography; and
- `US-simulation` for Ultra-Sonography.

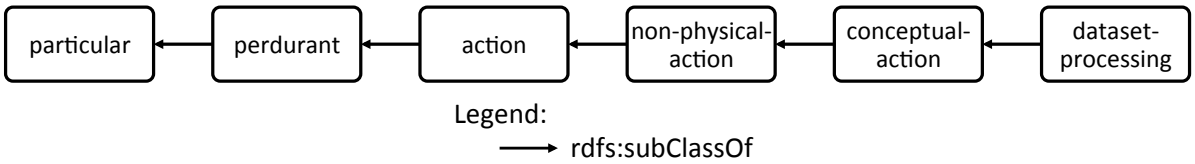


Figure 5.3: OntoVIP Excerpt - Dataset Processing Ascendance

The complete ascendance of `dataset-processing`, shown on Figure 5.3, all the way to generic notions of `perdurant` and `particular`, is an example of OntoVIP's reliance on DOLCE [Gangemi 02] and its foundational approach.

5.2.2 Workflow Designer

The `cowork` prototype has been slightly modified and integrated, packaged into a Java applet, into the VIP platform[11] as a "*Workflow Designer*": a module meant to help platform administrators design scientific workflows for new simulators to integrate into the platform.

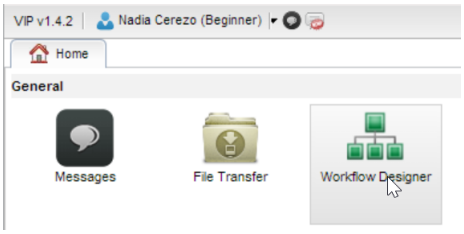


Figure 5.4: Link to VIP Workflow Designer Screenshot

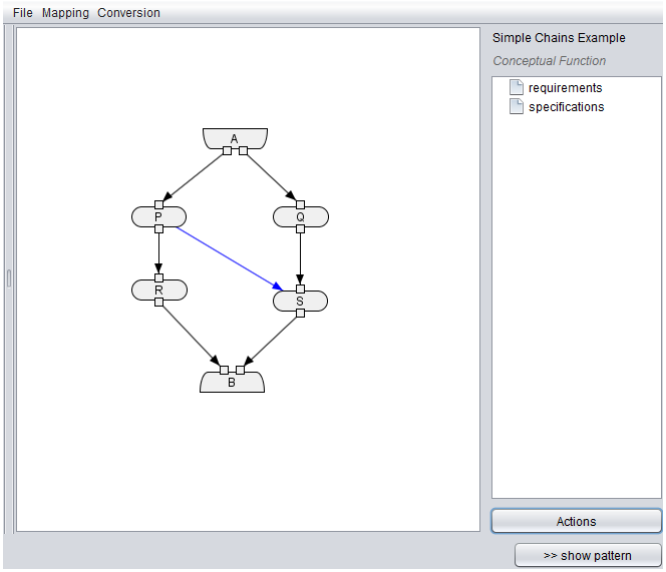


Figure 5.5: VIP Workflow Designer Screenshot

Figure 5.4 is a screenshot of the link to the Workflow Designer on the VIP platform, which is only visible to members of the `Cowork` group, and Figure 5.5 is a screenshot of the Workflow Designer interface, *i.e.* the prototype GUI.

[11] Virtual Imaging Platform: `http://vip.creatis.insa-lyon.fr/`

5.3 Conceptual Workflow Model

Our validation of the **Conceptual Workflow Model** is two-fold: on the one hand, we needed to make sure that it can model real-life scientific workflows; on the other hand, we needed to check that it does provide additional benefits, compared to purely abstract workflows.

Section 5.3.1 describes five of the simulators integrated into the VIP platform and shows the associated abstract workflows in GWENDIA [Montagnat 09], the language of the platform itself; Section 5.3.2 highlights a limitation of abstract workflows that our **Conceptual Workflow Model** alleviates; and Section 5.3.3 details five **Conceptual Workflows**, each modeling one of the five simulators.

5.3.1 VIP Simulators

Four simulators were seminally included from the start of the VIP project, to represent the four distinct modalities modeled in OntoVIP:

- FIELD-II [Jensen 04] for `US-simulation`;
- SIMRI [Benoit-Cattin 05] for `MR-simulation`;
- Sindbad [Tabary 09] for `CT-simulation`; and
- SORTEO [Reilhac 04] for `PET-simulation`.

Another simulator, SimuBloch [Cao 12], was integrated into the platform soon after launch and it quickly became popular among platform users for MR image simulation.

5.3.1.1 FIELD-II

FIELD-II [Jensen 04] is a Matlab[12] Application Programming Interface (API) developed by Jørgen Arendt Jensen:

> “ *Field II is a program for simulating ultrasound tranducer fields and ultrasound imaging using linear acoustics. The programs uses the Tupholme-Stepanishen method for calculating pulsed ultrasound fields. The program is capable of calculating the emitted and pulse-echo fields for both the pulsed and continuous wave case for a large number of different transducers.*
>
> OFFICIAL WEBSITE: `HTTP://FIELD-II.DK/`

In the context of VIP, FIELD-II [Jensen 04] is used to simulate medical UltraSonography, whose most well-known application is arguably obstetric sonography, commonly used during pregnancy.

[12]Matlab: `http://www.mathworks.com/products/matlab/`

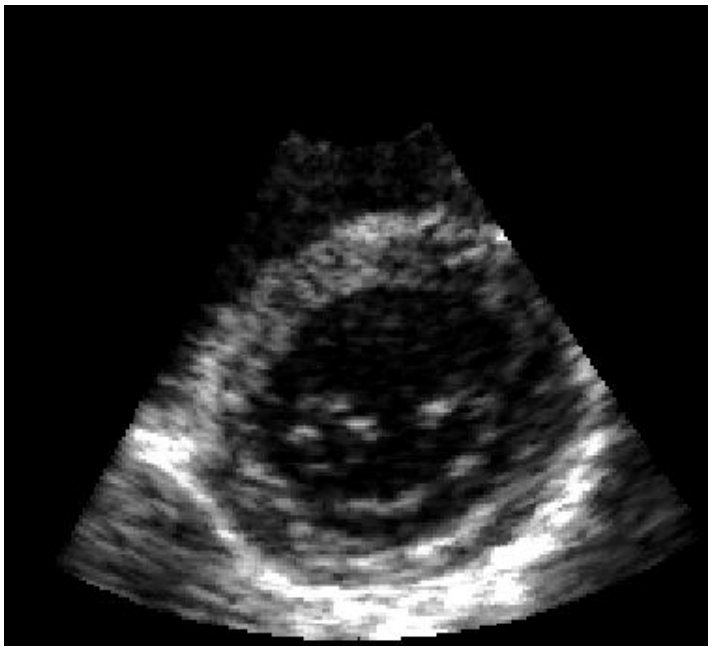


Figure 5.6: FIELD-II - Result Example

Figure 5.6 is one frame from a "*2D+t echocardiographic sequence [which] was simulated with an image-based approach*", taken from the VIP US Gallery[13] where more details about the simulation and contact information can be found.

The FIELD-II abstract workflow, which is shown on Figure 5.7, takes two Matlab files as input: `ProbeParameters_mat` contains all the simulation parameters, concerning the ultrasound probe (*e.g.* the elevation focus of the transducer) and the generated image (*e.g.* the number of lines in the image); and `TissueParameters_mat` contains the objet model, *i.e.* a structure of scatterers used to simulate a scanned object.

Data is split along result image lines and the split is handled by the main simulation step, `SimulateRFLine`: if the number of lines, computed by `read line number`, is less than the maximum number of jobs computed by `set max job number` based on the `simulation size` input, then one instance is invoked for each line. Otherwise, each instance of `SimulateRFLine` determines which lines it should process, based on a job number and the total number of jobs (*i.e.* the maximum number of jobs).

Simulated lines are merged into the output synthetic images of various formats (`RFImage`, `matImage`, `mhd` and `raw`) by `Merge` then `reconstructImage`. `Merge` is not connected to `SimulateRFLine` by either data or order link, but it is told how many lines should be merged and can thus know when they have all been processed. The reason for that peculiar structural choice is performance: it is a lot more efficient to merge the results of `SimulateRFLine` instances as they become available, rather than wait for all of them to finish to start. Moreover, `sleep` ensures that `Merge` starts before `SimulateRFLine`, to prevent a possible scheduling of all or most instances of `SimulateRFLine` before `Merge`.

[13] VIP US Gallery: `http://vip.creatis.insa-lyon.fr/gallery/us.html`

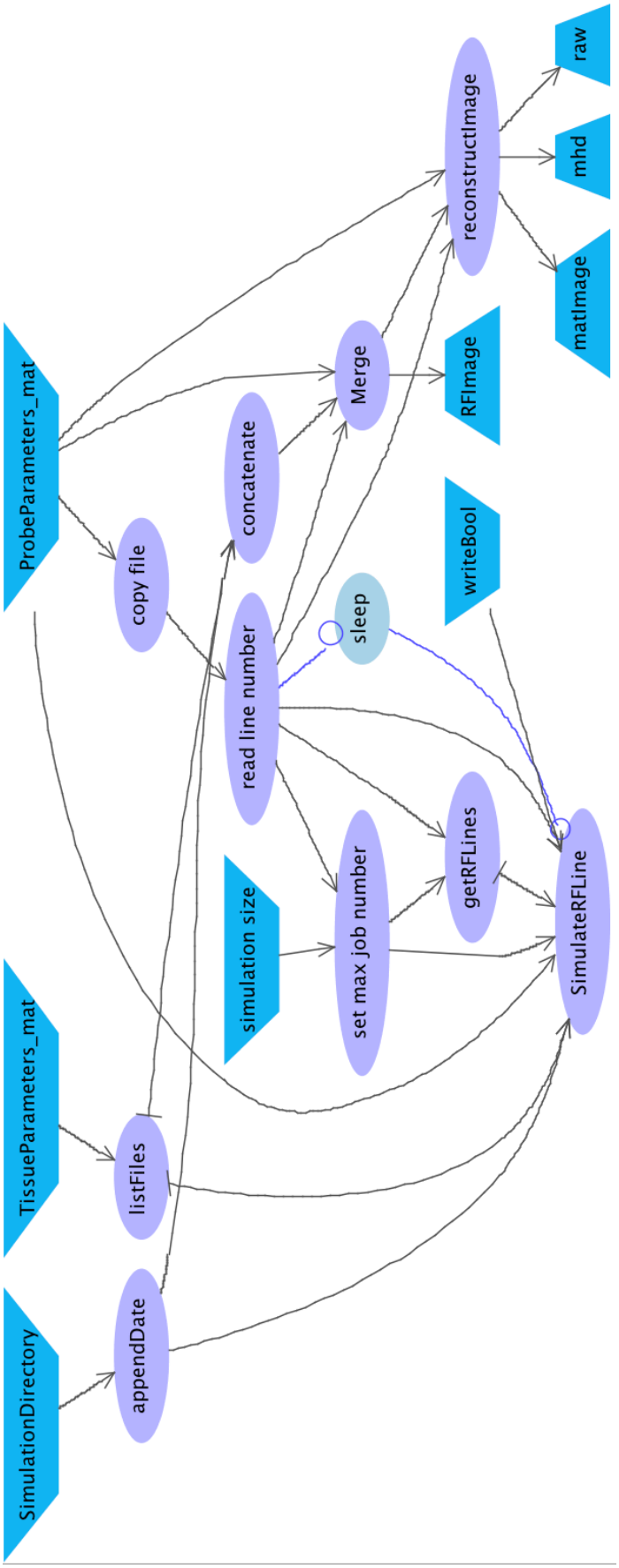


Figure 5.7: FIELD-II - Abstract Workflow (MOTEUR screenshot)

5.3.1.2 SIMRI

SIMRI [Benoit-Cattin 05] is developed at CREATIS[14] and distributed under the CeCiLL license[15]. As the name suggests, it simulates Magnetic Resonance Imaging (MRI), which uses magnetic fields to stimulate specific atoms into producing magnetic fields of their own, which are then detected by the scanner and used to construct a three-dimensional image. MRI contrasts soft tissues better than other medical imaging modalities, which makes it very useful in brain, muscles and heart imaging.

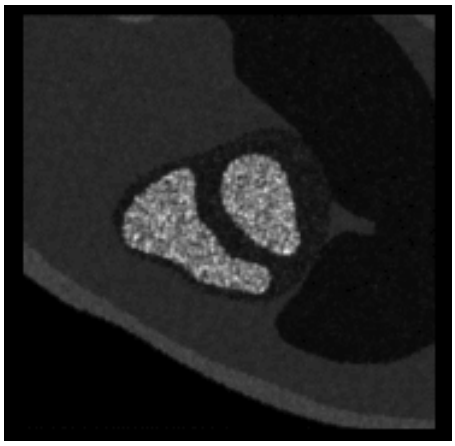


Figure 5.8: SIMRI - Result Example

Figure 5.8 is one instant of a "*Parasternal short axis view*" sequence, taken from the VIP MRI Gallery[16] where more details about the simulation and contact information can be found.

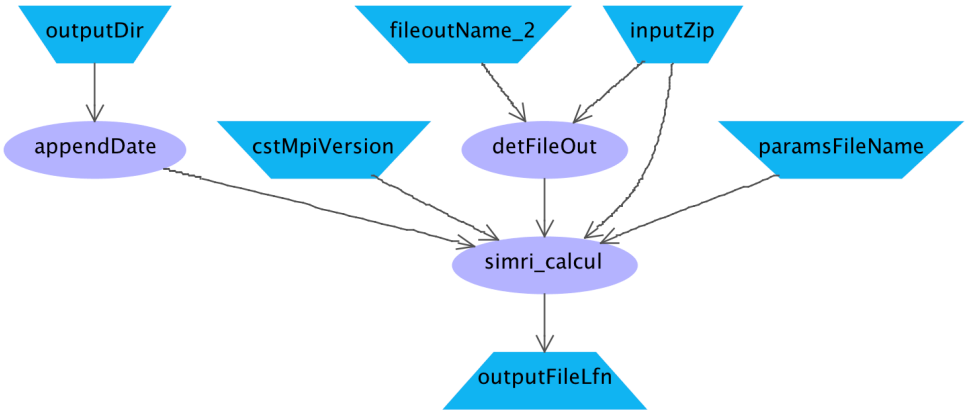


Figure 5.9: SIMRI - Abstract Workflow (MOTEUR screenshot)

The SIMRI abstract workflow, shown on Figure 5.9, has two main inputs: `inputZip` contains the object model and `paramsFileName` is the name of the file containing simulation parameters.

Because SIMRI was designed for deployment on parallel computing resources, the main program was parallelized using an MPI library [Benoit-Cattin 03]. As a result, the main step `simri_calcul` splits and merges data internally and then stores it on the grid and outputs the corresponding Logical File Name (LFN).

[14] CREATIS: `http://www.creatis.insa-lyon.fr/site/`
[15] CeCiLL license: `http://www.cecill.info/index.en.html`
[16] VIP MRI Gallery: `http://vip.creatis.insa-lyon.fr/gallery/mri.html`

5.3.1.3 SimuBloch

SimuBloch [Cao 12] is a Magnetic Resonance Imaging (MRI) simulator developed at VISAGES[17] which has quickly become popular among users of the VIP platform.

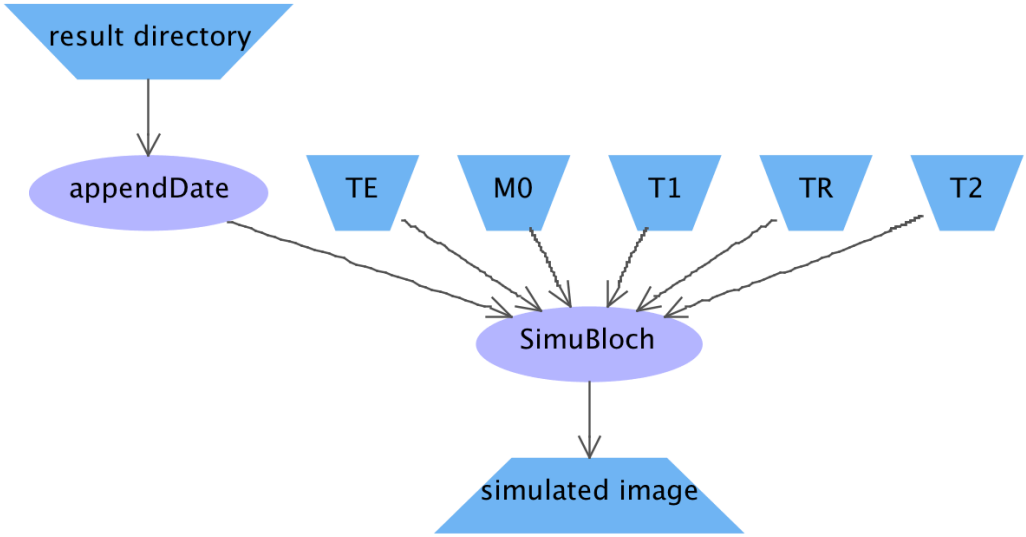


Figure 5.10: SimuBloch - Abstract Workflow (MOTEUR screenshot)

The SimuBloch abstract workflow, shown on Figure 5.10, has five main inputs. Three parameters compose the object model: `T1` represents the longitudinal magnetic relaxation time, `T2` represents the transverse relaxation time and `M0` represents the proton density. Two scalar values parameterize the simulation: `TR` is the repetition time and `TE` the echo time.

Like SIMRI, SimuBloch was designed for deployment on parallel computing resources and thus the main step `SimuBloch` leverages data parallelism internally and outputs the URL where it stored the final result.

5.3.1.4 Sindbad



Figure 5.12: Sindbad - Result Example

[17] VISAGES: `https://www.irisa.fr/visages/`

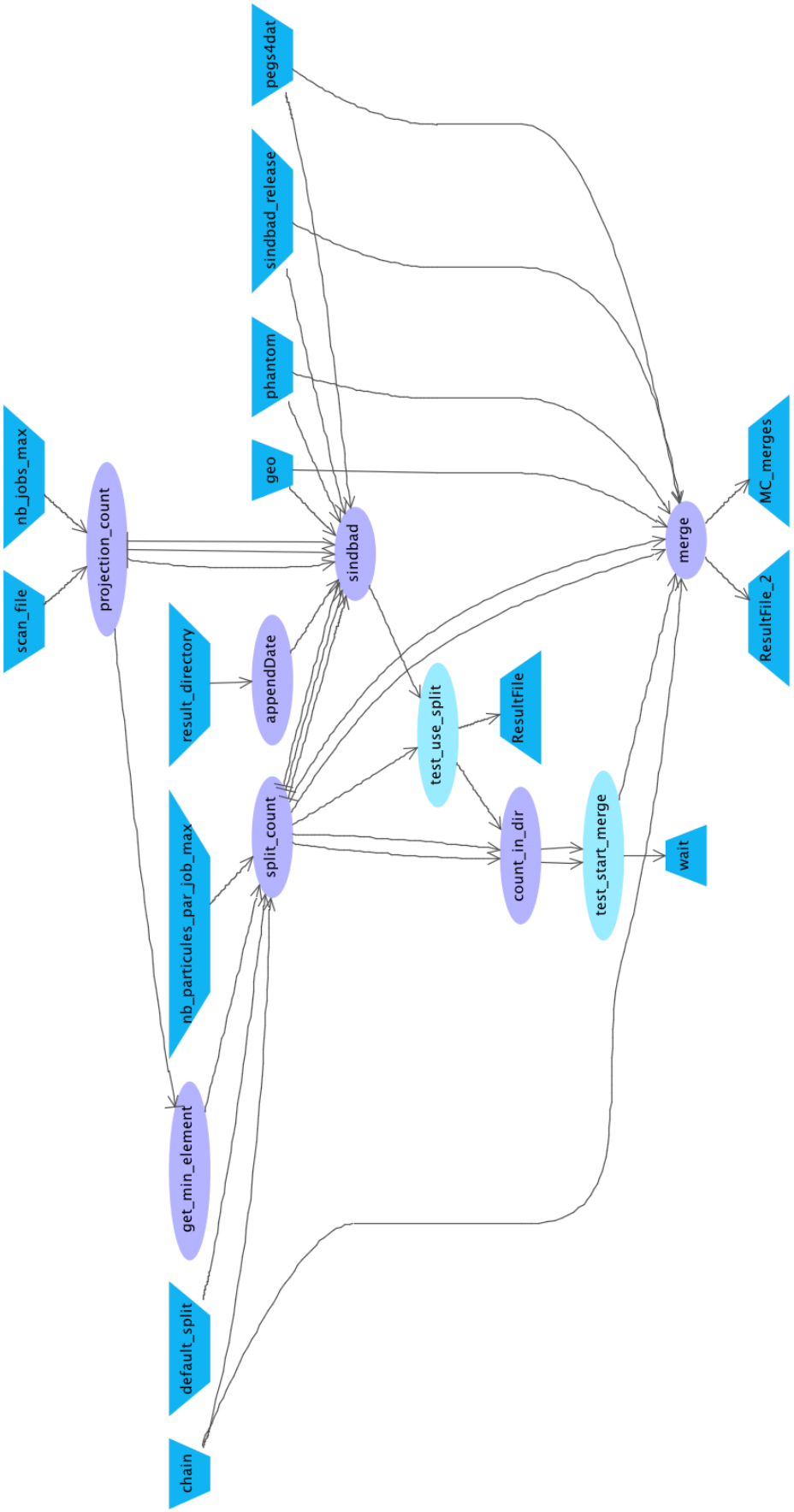


Figure 5.11: Sindbad - Abstract Workflow (MOTEUR screenshot)

Sindbad [Tabary 09] was developed at CEA[18] to simulate X-ray computed tomography (CT). CT can be construed as a three-dimensional version of traditional radiography: instead of taking X-ray images from only one angle, X-ray tomographs take images around an axis of rotation and those images are processed to generate a three-dimensional image.

Figure 5.12 is a "*Simulated whole-body static 3D CT acquisition (coronal slice)*" taken from the VIP CT Gallery[19] where more details about the simulation and contact information can be found.

The Sindbad abstract workflow, shown on Figure 5.11, takes one input for the object model, `phantom` and four for the simulation parameters: `pegs4dat` for electromagnetic properties of materials; `scan_file` for the scanner specifications; `geo` for geometric parameters; and `chain` for other simulation parameters.

A Sindbad simulation comprises a large number of projections, each of them resulting from the merging of an analysis and a Monte-Carlo computation. There are three levels of inherent data parallelism:

- projections (i.e. the individual 2D images) are independent;
- analysis and processing of each projection can be done in parallel; and
- random samplings of each Monte-Carlo algorithm can be computed simultaneously.

Much like with FIELD-II, data splitting is handled by the main step `sindbad`, however the test of whether merging can start is done explicitly with conditionals rather than inside `merge`.

5.3.1.5 SORTEO

SORTEO [Reilhac 04] (Simulation Of Realistic Tridimensional Emitting Objects) is a simulation software built at the CERMEP imaging centre, Lyon, France and the McConnell Brain Imaging Centre, Montreal, Canada. It uses Monte-Carlo techniques to simulate Positron Emission Tomography (PET): tracers (radioactive atoms) are introduced in the body of the patient and the gamma rays they indirectly emit are detected in all directions by a tomograph and analyzed to build a tridimensional image of the tracer concentration.

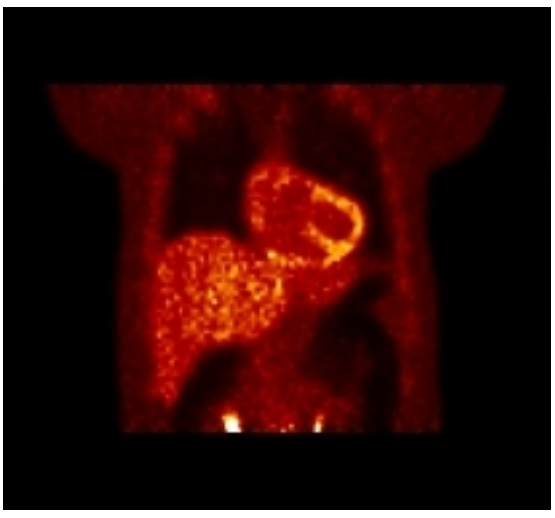


Figure 5.13: SORTEO - Result Example

[18] CEA: `http://www.cea.fr/`
[19] VIP CT Gallery: `http://vip.creatis.insa-lyon.fr/gallery/ct.html`

Figure 5.13 is a "*Simulated whole-body static 3D 224s FDG-PET acquisition (coronal slice)*" taken from the VIP PET Gallery[20] where more details about the simulation and contact information can be found.

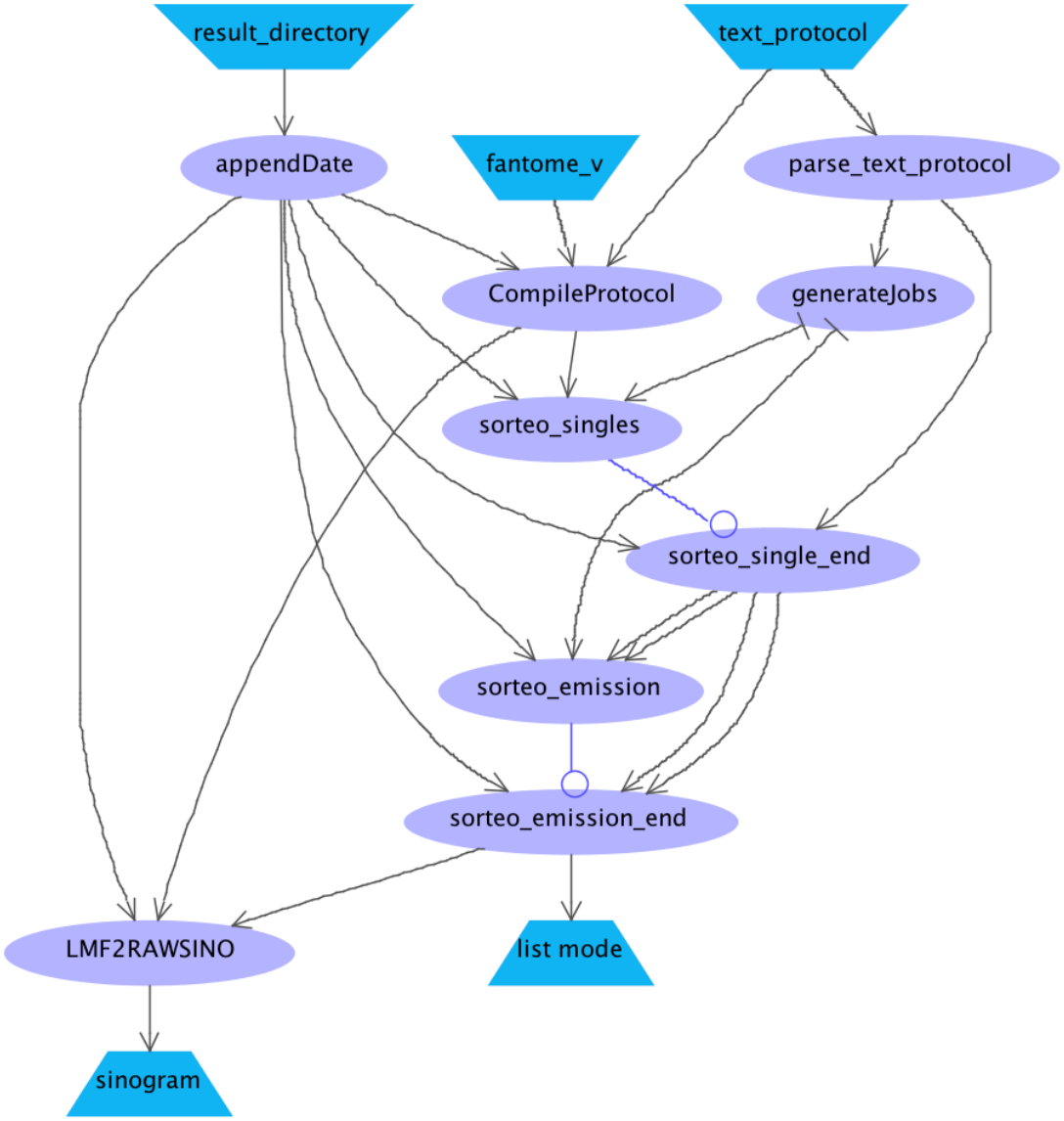


Figure 5.14: SORTEO - Abstract Workflow (MOTEUR screenshot)

The SORTEO abstract workflow, shown on Figure 5.14, simulates a the PET procedure in two separate steps:

- first singles (short for single photons) are generated with `sorteo_singles` and the resulting data is merged by `sorteo_single_end`,
- then the resulting positron emissions are computed with `sorteo_emission` and the resulting data is merged by `sorteo_emission_end`.

Data is split only once by `generateJobs`. The object model is contained in a file called `fantome_v` and the simulation parameters in `text_protocol`. All other **Abstract Elements** pertain to technicalities, *e.g.* `LMF2RAWSINO` converts from the list mode format result of SORTEO to a raw sinogram.

[20] VIP PET Gallery: `http://vip.creatis.insa-lyon.fr/gallery/pet.html`

5.3.2 Simulator Template

All five aforementioned VIP simulators share a common "*Simulator Template*". They all fit the same high-level basic skeleton of:

- two types of input data:
  - data that defines the `Object Model`, *i.e.* the anatomical part and/or pathology whose medical image will be synthesized, *e.g.* a brain voxel; and
  - `Simulation Parameters` defining the medical imaging procedure which will be simulated, *e.g.* location of the scanner;
- a main process step, `Simulate Medical Imaging Procedure`; and
- output data of only one type, `Synthetic Medical Image`.

Moreover, in the context of VIP, all four simulators leverage data parallelism: a **Concern** we called `SplitAndMerge` because fulfilling it consists in splitting data to spread it over multiple processing units and then merging the results.

Since the OntoVIP ontology does not model **Concerns** like `SplitAndMerge`, we created a small taxonomy `cowork-annot.rdfs` to supplement OntoVIP with precisely the `SplitAndMerge` **Concern** and the related `Split` and `Merge` **Functions**.

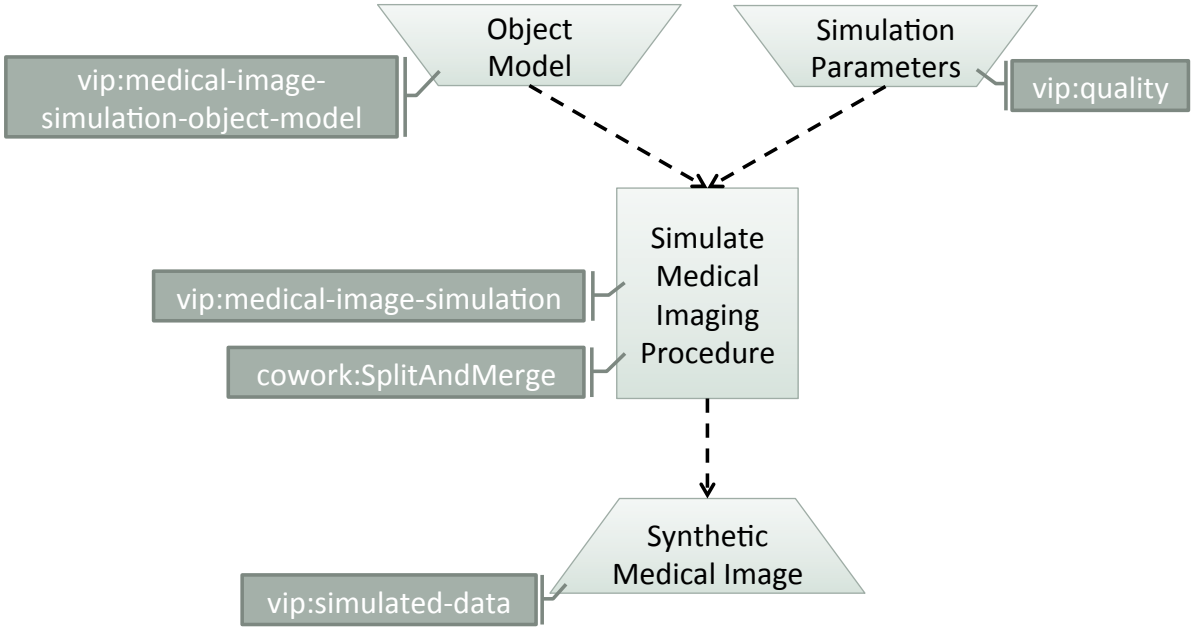


Figure 5.15: VIP Simulator Template Conceptual Workflow

That common template, modeled as a **Conceptual Workflow** on Figure 5.15, is near impossible to detect in the abstract workflows and cannot be modeled in a purely abstract scientific workflow model such as GWENDIA [Montagnat 09]. Conversely, since **Intermediate Representations** present both the **Conceptual Level** and **Abstract Level** of a scientific workflow, they can fully model the five simulators and tie them to the high-level template.

5.3.3 Conceptual Workflows

In the following five **Conceptual Workflows**, which we designed manually to check the expressivity of our model: (i) **Input Ports** are annotated like the **Output Port** they are bound to by a **Data Link** (ii) the `SplitAndMerge` **Concern** as well as the `Split` and `Merge` **Functions** come from the `cowork-annot` taxonomy we created to supplement OntoVIP and (iii) all other **Annotations** come from the modules of OntoVIP: the full URIs can be found in Appendix G.

In most of the following **Conceptual Workflow** figures, we used a graphical tool we called the **Data Link Replicator**: it splits **Data Links** which would otherwise cross-cut the graph and make it less legible.

5.3.3.1 FIELD-II

The FIELD-II workflow fits the template with an additional `Merge` **Conceptual Function**, while the correspondig `Split` **Function** is catered to internally in the `SimulateRFLine` **Activity**.

The vast majority of the **Activities** featured in the workflow cater to technical issues and thus do not belong to any **Conceptual Function**, *e.g.* `read line number` which extracts a specific information from the input parameter `ProbeParameters_mat` for other **Activities** to process and `reconstructImage` which converts the result into additional output formats.

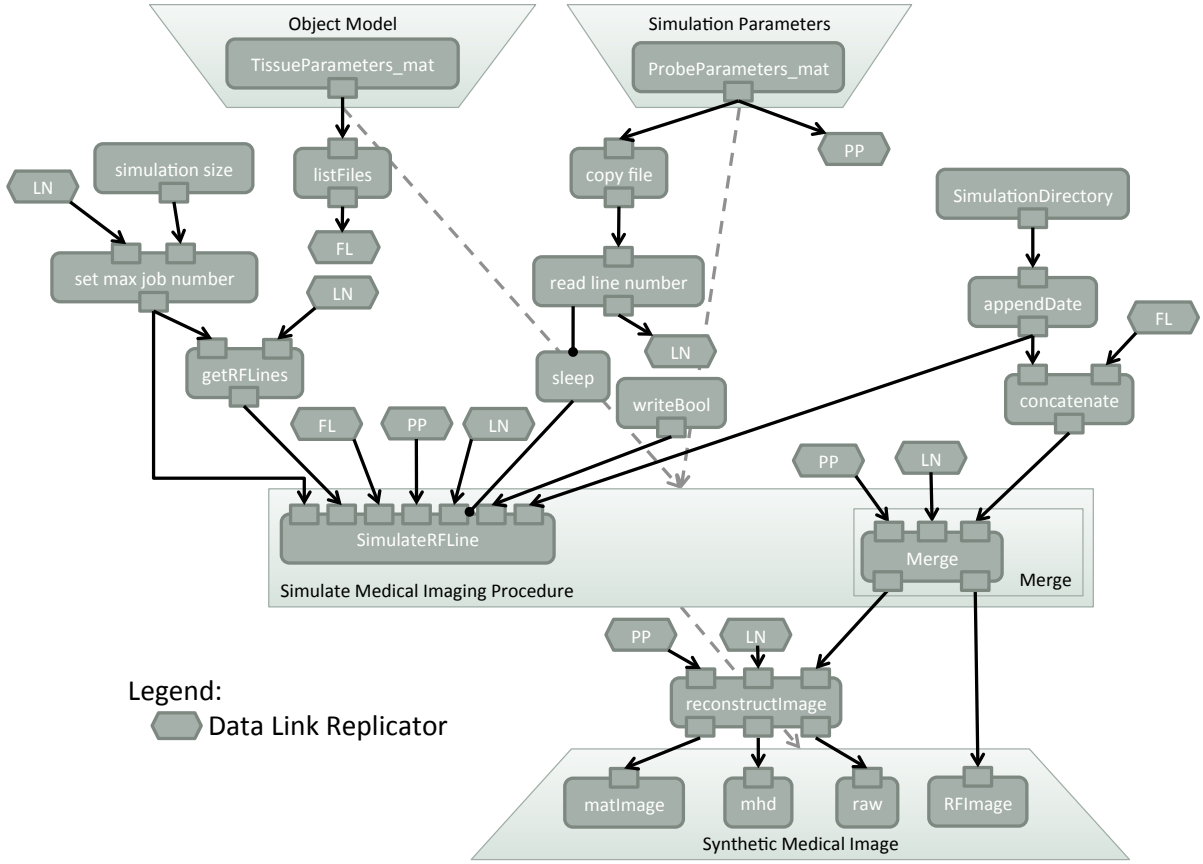


Figure 5.16: FIELD-II Conceptual Workflow

The elements of the **Intermediate Representation** of FIELD-II, shown on Figure 5.16, are annotated with the following **Specifications**:

- `Object Model` and the **Output Port** of `TissueParameters_mat` bear `US-simulation-compatible-model`;
- `Simulation Parameters` and the **Output Ports** of `ProbeParameters_mat`, `copy file` and `simulation size` bear `quality`;
- `Simulate Ultra-Sonography` and `SimulateRFLine` perform `US-simulation` and `Split`;
- the `Merge` **Conceptual Function** and `activity` both perform `Merge`;
- `Synthetic Sonogram` and the **Output Ports** of `Merge` and `reconstructImage` bear `US-simulated-image`;
- the **Output Ports** of `SimulationDirectory` and `appendDate` bear `directory`;
- the **Output Ports** of `listFiles` and `concatenate` bear `file`; and
- the **Output Ports** of `read line number`, `getRFLines` and `set max job number` bear `number`.

5.3.3.2 SIMRI

The SIMRI workflow fits the template almost directly. It only featues five **Activities** (including two **Inputs**) which do not pertain to any **Conceptual Function**.

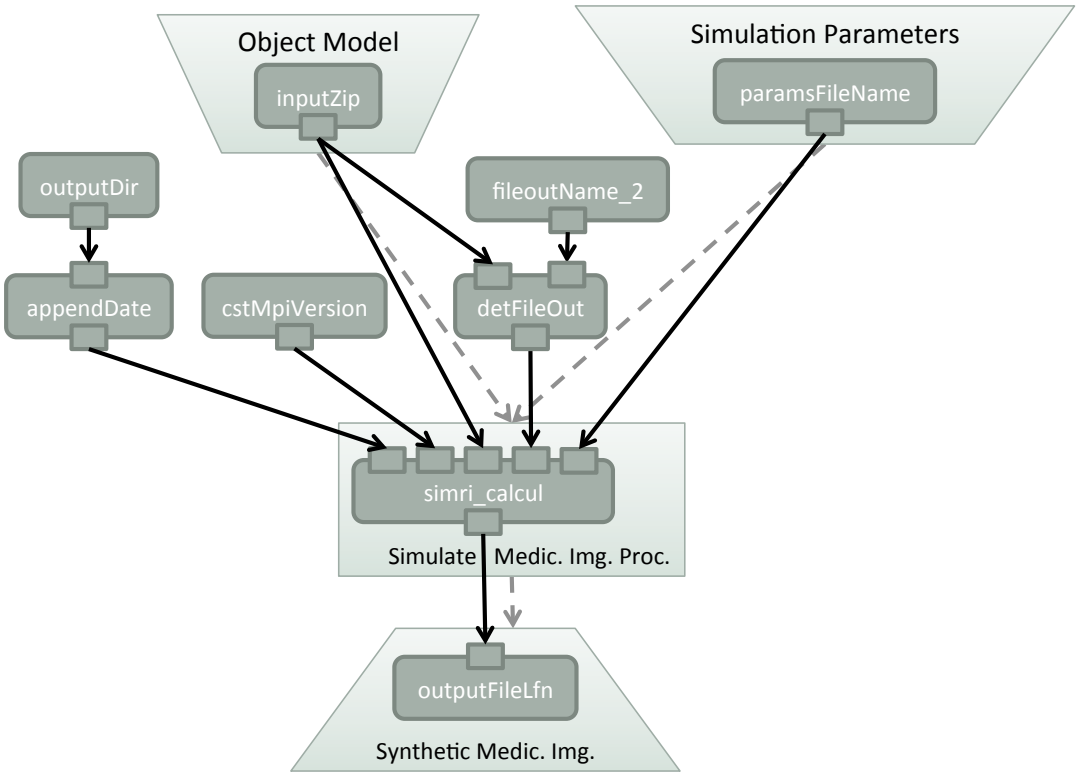


Figure 5.17: SIMRI Conceptual Workflow

The elements of the **Intermediate Representation** of SIMRI, shown on Figure 5.17, are annotated with the following **Specifications**:

- `Simulate MRI` and `SIMRI` perform `ontovip:MR-simulation` and fulfill `SplitAndMerge`;
- `Simulation Parameters` and the **Output Port** of `paramsFileName` bear `quality`;
- `Object Model` and the **Output Port** of `inputZip` bear `MR-simulation-compatible-model`;
- `Synthetic MRI` and the **Output Port** of `simri_calcul` bear `MR-simulated-image`;
- the **Output Ports** of `outputDir` and `appendDate` bear `directory`;
- the **Output Port** of `cstMpiVersion` bears `version-number`; and
- the **Output Ports** of `fileoutName_2` and `detFileOut` bear `file`.

5.3.3.3 SimuBloch

Of the five simulators, `SimuBloch` is the one whose implementation as an abstract workflow sticks closest to the simulator template. It only features two technical **Activities**, `result directory` and `appendDate`, to generate the result directory name in a manner compliant with platform conventions.

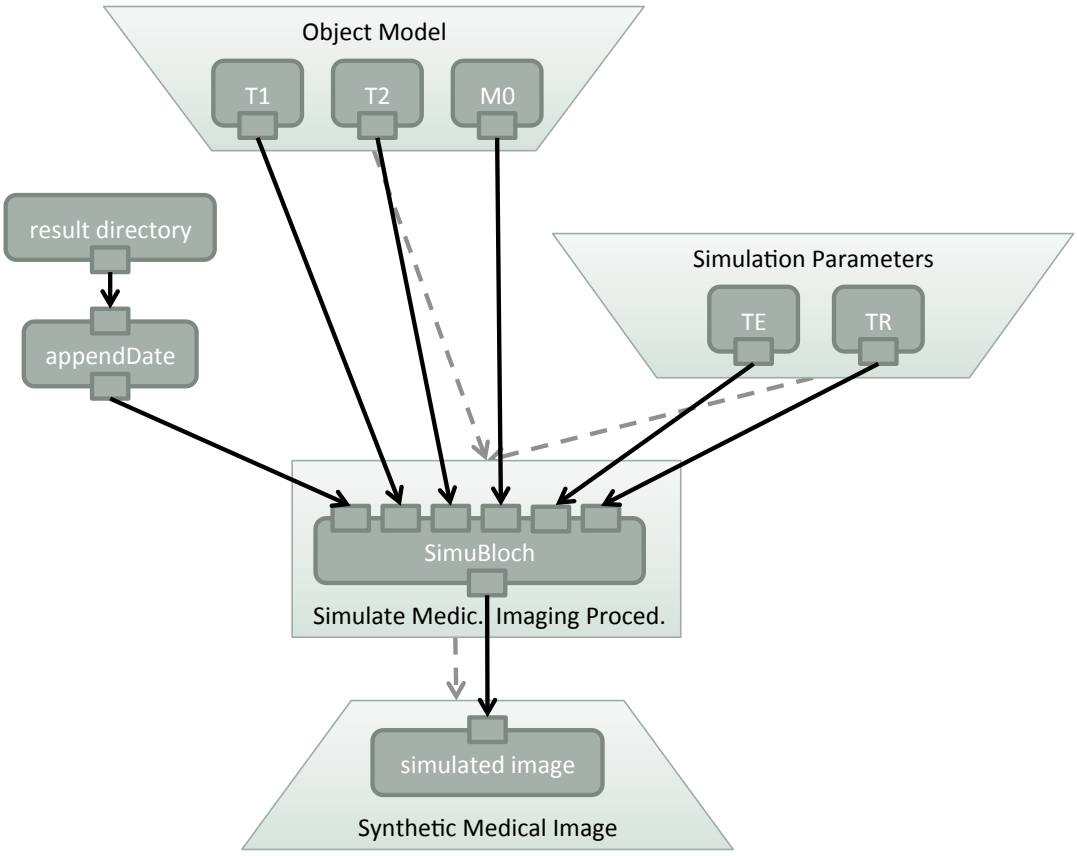


Figure 5.18: SimuBloch Conceptual Workflow

The elements of the **Intermediate Representation** of SimuBloch, shown on Figure 5.18, are annotated with the following **Specifications**:

- `Simulate MRI` and `SimuBloch` perform `ontovip:MR-simulation` and fulfill `SplitAndMerge`;
- `Simulation Parameters` bears `quality`;
- `Object Model` bears `MR-simulation-compatible-model`;
- `Synthetic MRI` and the **Output Port** of `SimuBloch` bear `MR-simulated-image`;
- the **Output Ports** of `result directory` and `appendDate` bear `directory`;
- the **Output Port** of `T1` bears `T1-weighted-MR-dataset`;
- the **Output Port** of `T2` bears `T2-weighted-MR-dataset`;
- the **Output Port** of `M0` bears `proton-density-weighted-MR-dataset`;
- the **Output Port** of `TE` bears `echo-time`; and
- the **Output Port** of `TR` bears `repetition-time`.

5.3.3.4 Sindbad

The `Sindbad` workflow, like that of `FIELD-II`, fits the template with an additional `Merge` **Conceptual Function** while the `Split` **Function** is performed by the `sindbad` **Activity** and about half of its **Activities** pertain to no **Conceptual Function**.

The elements of the **Intermediate Representation** of Sindbad, shown on Figure 5.19, are annotated with the following **Specifications**:

- `Object Model` and the **Output Port** of `phantom` bear `CT-simulation-compatible-model`;
- `Simulation Parameters` and the **Output Ports** of `chain`, `geo`, `scan_file`, `pegs4dat`, `split_count` and `projection_count` bear `quality`;
- the **Output Ports** of `nb_jobs_max`, `get_min_element` and `default_split` bear `number`;
- the **Output Port** of `sindbad_release` bears `version-number`;
- `Synthetic CT Scan` and the **Output Ports** of `sindbad`, `test_use_split`, `count_in_dir`, `text_start_merge` and `merge` bear `CT-simulated-data`;
- `Simulate CT Scan` and `sindbad` perform `CT-simulation` and `Split`; and
- the `Merge` **Conceptual Function** and the `merge` **Activity** both perform `Merge`.

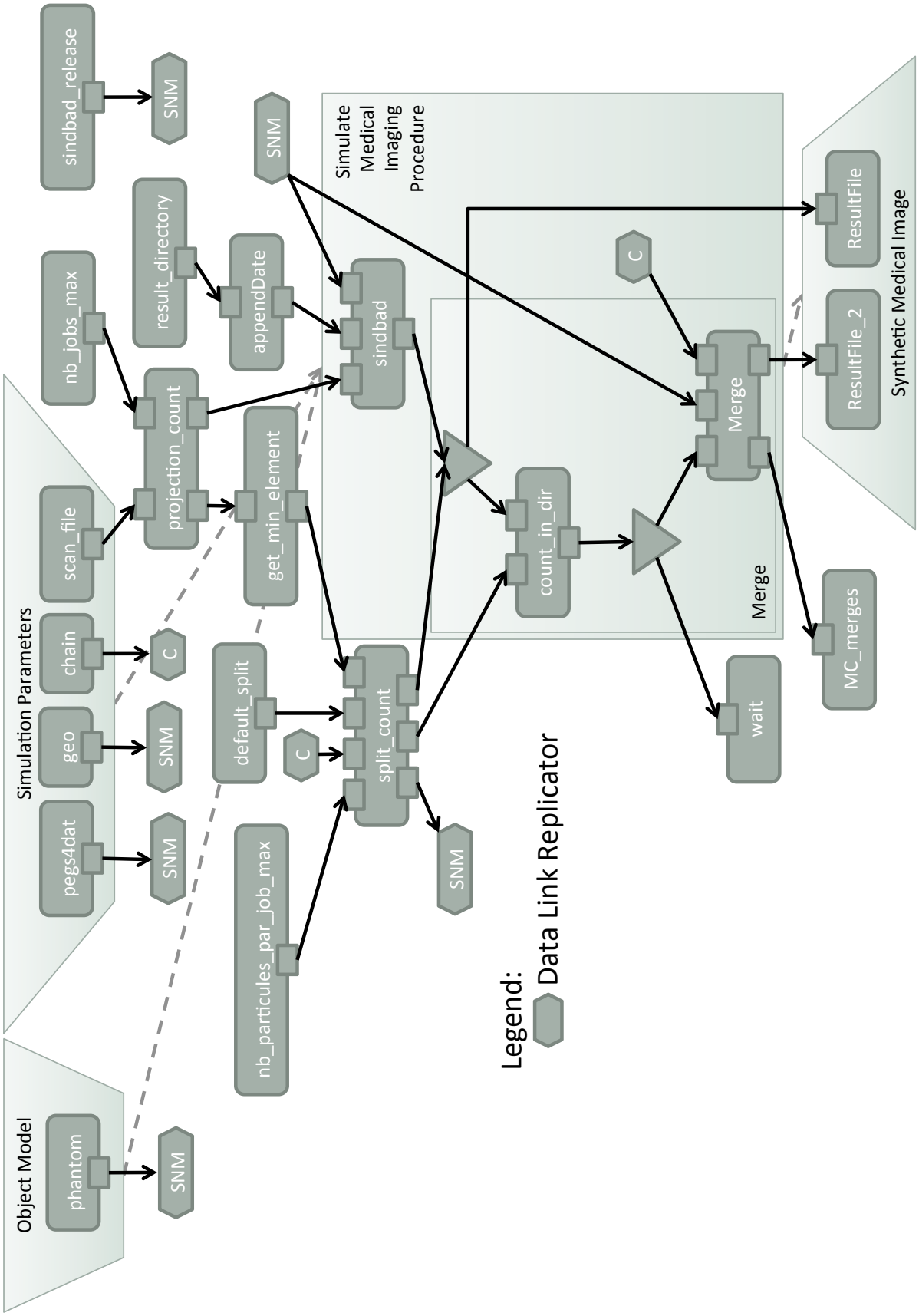


Figure 5.19: Sindbad Conceptual Workflow

5.3.3.5 SORTEO

The relationship with the template is most complex with the SORTEO abstract workflow. Indeed, not only does it feature additional `Split` and `Merge` **Conceptual Functions** to handle the `SplitAndMerge` **Concern**, it also splits the main **Conceptual Function** into two distinct phases `Generate Singles` and `Generate Emissions`. Of course, it also features a handful of technical **Activities** belonging to no **Conceptual Function** in particular.

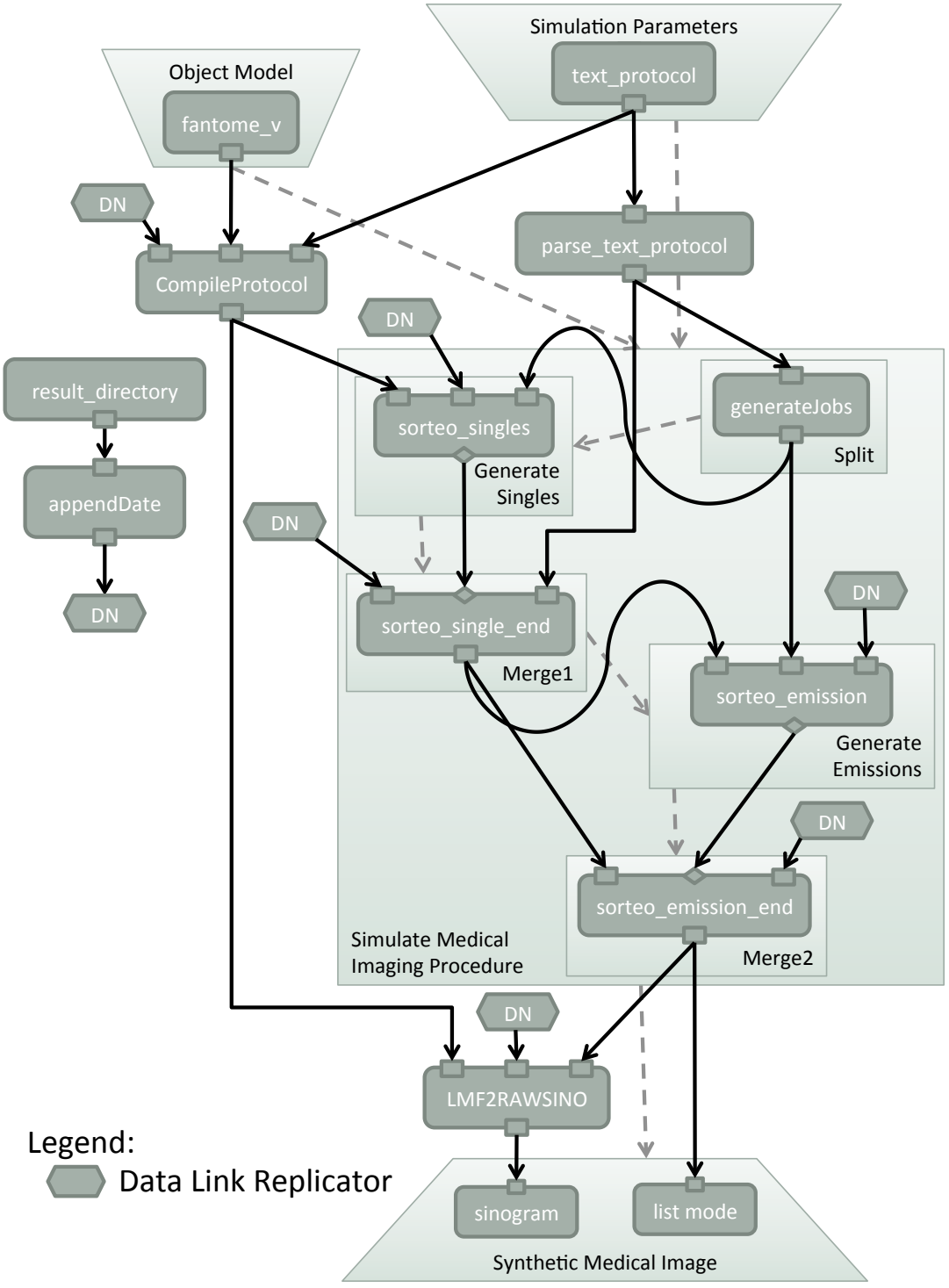


Figure 5.20: SORTEO Conceptual Workflow

The elements of the **Intermediate Representation** of SORTEO, shown on Figure 5.20, are annotated with the following **Specifications**:

- `Generate Singles`, `Generate Emissions`, `sorteo_singles` and `sorteo_emission` perform `PET-simulation`;
- `Split` and `generateJobs` perform `split`;
- `Merge1`, `Merge2`, `sorteo_single_end` and `sorteo_emission_end` perform `Merge`;
- `Simulation Parameters` and the **Output Ports** of `text_protocol`, `parse_text_protocol`, `generateJobs` and `CompileProtocol` bear `quality`;
- `Object Model` and the **Output Port** of `fantome_v` bear `PET-simulation-compatible-model`;
- `Synthetic PET Sinogram` bears `PET-simulated-image`;
- the **Output Ports** of `result_directory` and `appendDate` bear `directory`;
- the **Output Port** of `sorteo_single_end` bears `PET-simulated-data`;
- the **Input Port** of `sorteo_emission_end` bears `PET-list-mode-data`; and
- the **Output Port** of `LMF2RAWSINO` bears `PET-sinogram`.

5.3.4 Discussion

Applying the **Conceptual Workflow Model** to the five VIP simulators has allowed us to check that it can model real simulations.

Moreover, it shows that **Conceptual Workflows**, thanks to their multiple abstraction levels and structural flexibility, can be used to:

- highlight common high-level templates among workflows whose implementations differ widely; and
- emphasize the simulation in a scientific workflow, making it easier to interpret by explicitiing the scientific goals.

However, **Conceptual Workflows** tend to take more graphical space than the equivalent abstract workflows, because of their additional layers of information, and that makes **Conceptual Workflows** of full-fledged simulations, like the VIP simulators presented here, hard to fit for print. We have tackled that issue by using a purely graphical shortcut we called `Data Link Replicator` which splits **Data Links** so they do not cross the rest of the workflow. It would be interesting to investigate whether it enhances or degrades legibility and whether its use can be automated.

5.4 Transformation Process

To validate our **Transformation Process**, we opted for the use case presented in Section 5.4.2, inspired by the VIP simulators, and applied on it the **Mapping** and **Conversion** processes. The results are detailed in Section 5.4.3 for the algorithms of **Discovery** and **Weaving**; in Section 5.4.4 for that of **Composition**; and in Section 5.4.5 for the **Conversion** to GWENDIA [Montagnat 09], which is used on the VIP platform.

All the algorithms of **Mapping** were run against the same knowledge base containing only the **Fragments** described in the following section.

5.4.1 VIP Fragments

The **Intermediate Representations** of the aforementioned simulators are wrapped into **Fragments** before they are saved into the knowledge base, but they are not easily woven as-is. Other **Fragments** can be extracted from them to facilitate reuse.

5.4.1.1 Simple Sub-workflows

The simplest form of **Fragment**, besides those with empty **Patterns**, is that of a single **Conceptual Workflow** fulfilling a set of **Requirements** by embedding a set of **Activities**.

The template is as follows: a **Pattern** containing a single **Conceptual Workflow** which bears a set of **Requirements** and a **Blueprint** containing the same (identified by name) **Conceptual Workflow**, with at least one of the aforementioned **Requirements** transformed into a **Specification**.

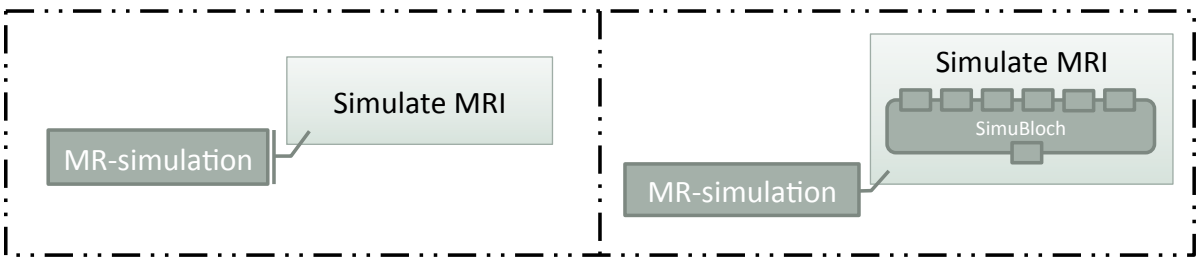


Figure 5.21: Simple Sub-workflow Fragment Example

For instance, the `SimuBloch` **Fragment**, shown on Figure 5.21, features a **Pattern** annotated with a `MR-simulation` **Requirement** and a **Blueprint** annotated with the same **Type**, but as a **Specification**. The basic meaning of this **Fragment** is that the **Activity** `SimuBloch` performs the `MR-simulation` **Function**.

5.4.1.2 Two Steps Function

A more complex type of **Fragment** aims to fulfill a **Requirement** not by a sub-workflow of **Activities**, but by a sub-workflow of **Conceptual Workflows**. That has multiple benefits, including the ability to introduce further **Requirements** for later **Discovery** steps. However, the **Fragment** designer must know more about the inner workings of the **Weaving** process, especially when it comes to binding (*cf.* Section 4.2.6): whether the generated elements are bound to nodes or to links will impact where and how many elements will be generated.

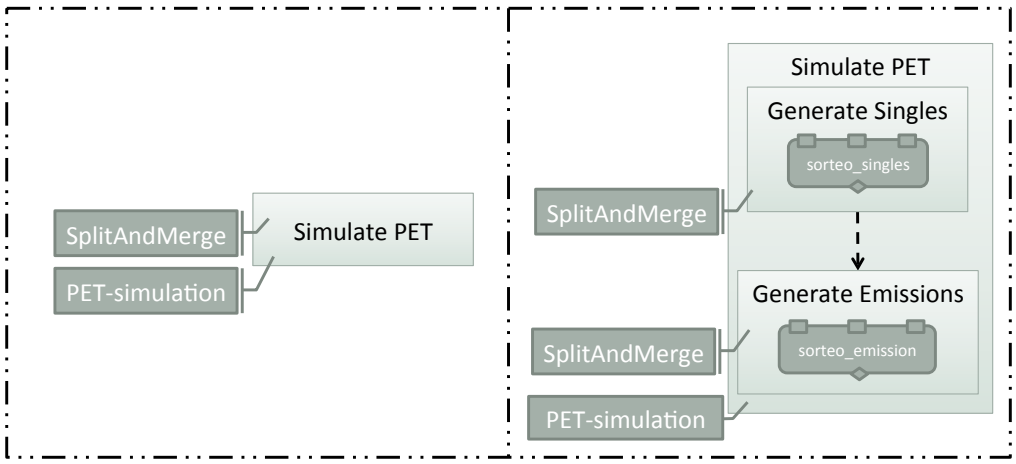


Figure 5.22: PET 2 Steps Fragment

For instance, the `PET 2 Steps` **Fragment**, shown on Figure 5.22, describes how SORTEO performs the `PET-simulation` **Function**: in two distinct phases, each of which must fulfill the `SplitAndMerge` **Concern** and it uses node-bound **Weaving** (*cf.* Section 4.2.6.1) to do so.

5.4.1.3 Split and Merge

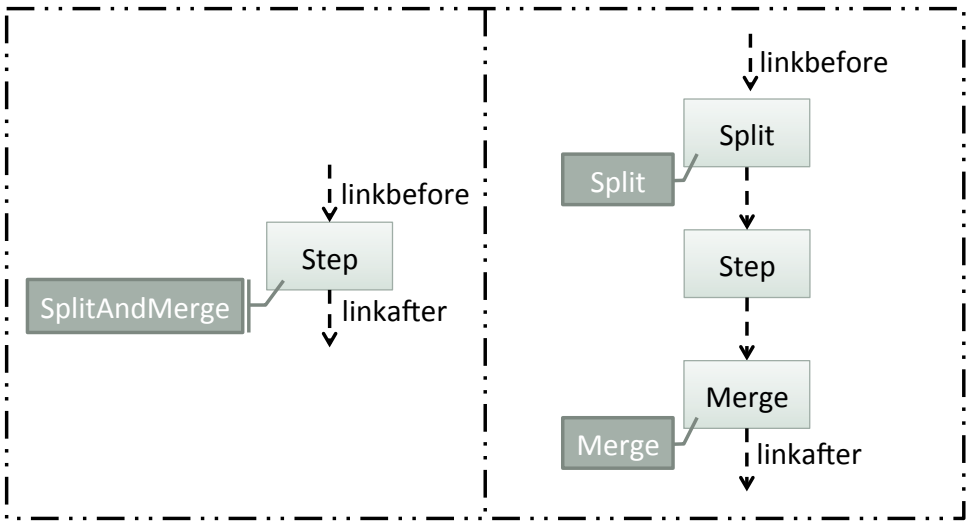


Figure 5.23: Split and Merge Fragment - Link-bound

The most complex **Fragment** that can be extracted from the VIP simulators is the `Split and Merge` **Fragment** shown on Figure 5.23 and Figure 5.24. The former uses link-bound and the latter node-bound **Weaving** (*cf.* Section 4.2.6.2 and Section 4.2.6.1 respectively) to insert a pre-processing `Split` step and a post-processing `Merge` step, in order to fulfill the `SplitAndMerge` **Concern**.

On the one hand, even though `FIELD-II`, `Sindbad` and `SORTEO` each fulfill the **Concern** in their own distinct way, all three methods fit in the high-level skeleton created by those **Fragments**. `SIMRI` and `SimuBloch`, on the other hand, fulfill the **Concern** inside their main **Activity** and thus require no additional **Conceptual Functions**.

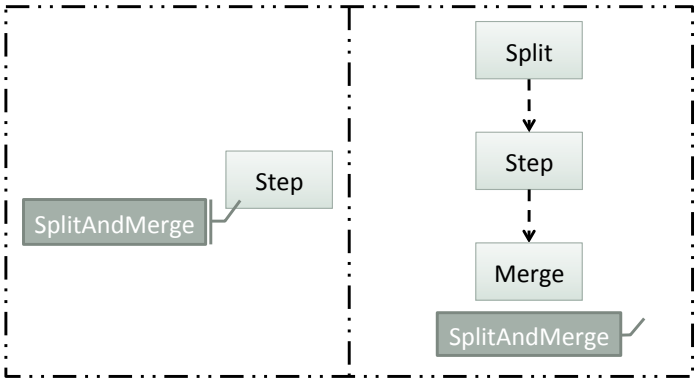


Figure 5.24: Split and Merge Fragment - Node-bound

5.4.2 Use Case

One of the goals of the VIP project was to ease the design and integration of new medical image simulation tools. To illustrate how **Conceptual Workflows** help reuse existing elements, let us consider the following situation: the knowledge base contains the five **Conceptual Workflows** and **Fragments** detailed in the previous Section 5.3.1 and we want to create a workflow to simulate MRI and PET scans of a given medical object simultaneously, similarly to what is done in hybrid PET/MRI image acquisition devices available today.

The first step is to design a high-level **Conceptual Workflow** which fits our needs. A simple variation of the "*VIP Simulator Template*" described in Section 5.3.2, as shown on Figure 5.25 will do nicely.

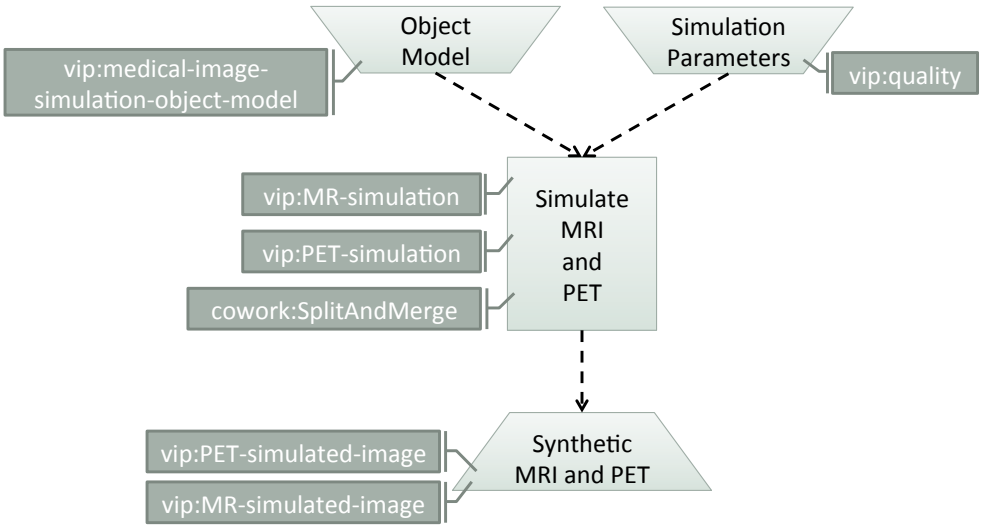


Figure 5.25: Use Case High-level Conceptual Workflow

The modifications required pertain to the **Annotations** which must let the system know the aim is to simulate both an MRI and a PET scan:

- the `vip:medical-image-simulation` **Function** is split into `vip:MR-simulation` and `vip:PET-simulation` instead; and
- the `vip:simulated-data` **Dataset** is split into `vip:MR-simulated-image` and `vip:PET-simulated-image`.

5.4.3 Discovery and Weaving

In this entire section, the following constants were used (*cf.* Section 4.4.3.2): exact matches constant $K_{EM} = 1.0$; narrower matches constant $K_{NM} = 0.5$; broader matches constant $K_{BM} = 0.25$; and functions priority factor $K_F = 2.0$.

Discovery applied to the `Simulate MRI and PET` **Conceptual Function** finds the following matches (Legend: `Fragment name` [*score*]):

1. `PET 2 Steps` [0.6]
2. `SimuBloch` [0.4]
3. `Split and Merge` [0.2]
4. `Node-bound Split and Merge` [0.2]
5. `SIMRI (complete workflow)` [0.15]
6. `SimuBloch (complete workflow)` [0.15]
7. `SORTEO (complete workflow)` [0.09]

The `PET 2 Steps` **Fragment** is first because it caters to both the `PET-simulation` and `SplitAndMerge` **Requirements**, but the score is not perfect since it does not perform `MR-simulation`. As expected, the difference in scores between the `SimuBloch` **Fragment** and the `Split and Merge` ones reflects exactly the functions priority factor. Complete workflows performing either `PET-simulation` or `MR-simulation` get a much lower score because of their extra **Functions**.

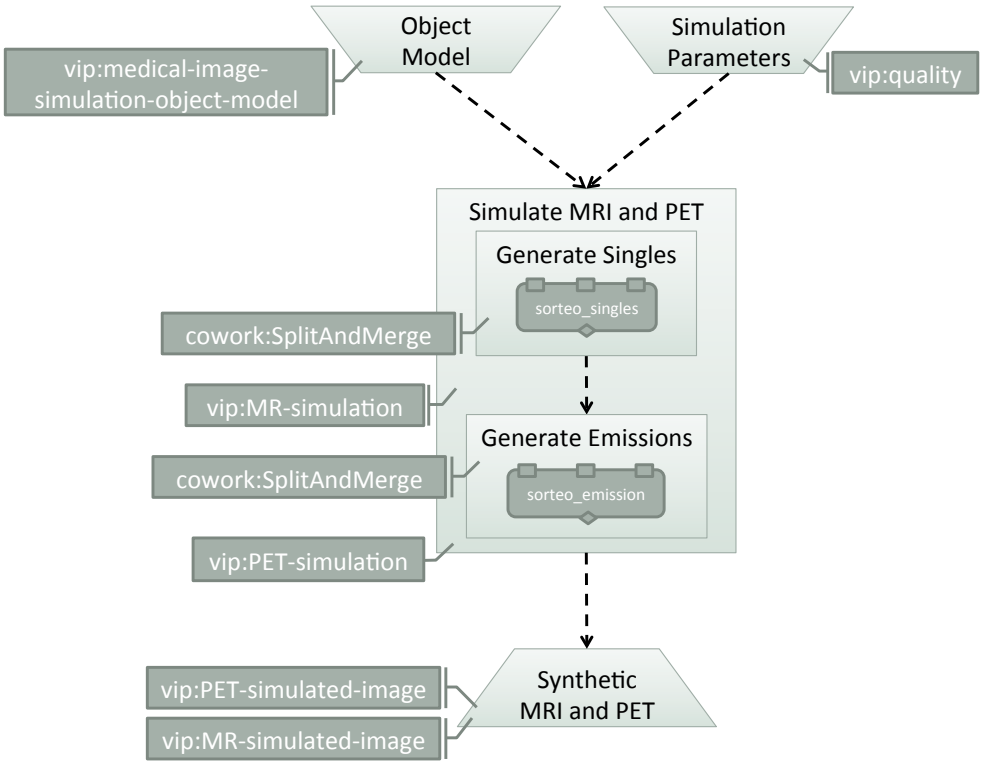


Figure 5.26: Mapping Use Case - After Weaving PET 2 Steps

Figure 5.26 shows the result of **Weaving** the top discovered match `PET 2 steps` (shown on Figure 5.22) into the base workflow (shown on Figure 5.25).

This time, applying **Discovery** to the `Simulate MRI and PET` **Conceptual Function** finds the following matches (Legend: `Fragment name` [*score*]):

1. `SimuBloch` [0.67]
2. `Split and Merge` [0.33]
3. `Node-bound Split and Merge` [0.33]
4. `SIMRI (complete workflow)` [0.25]
5. `SimuBloch (complete workflow)` [0.25]
6. `PET 2 Steps` [0.08]
7. `SORTEO (complete workflow)` [0.03]

Again, the difference in scores between the `SimuBloch` **Fragment** and the `Split and Merge` ones reflects the functions priority factor. Since `PET-simulation` is no longer a **Requirement**, the `PET 2 Steps` **Fragment** fell to the bottom of the ranking.

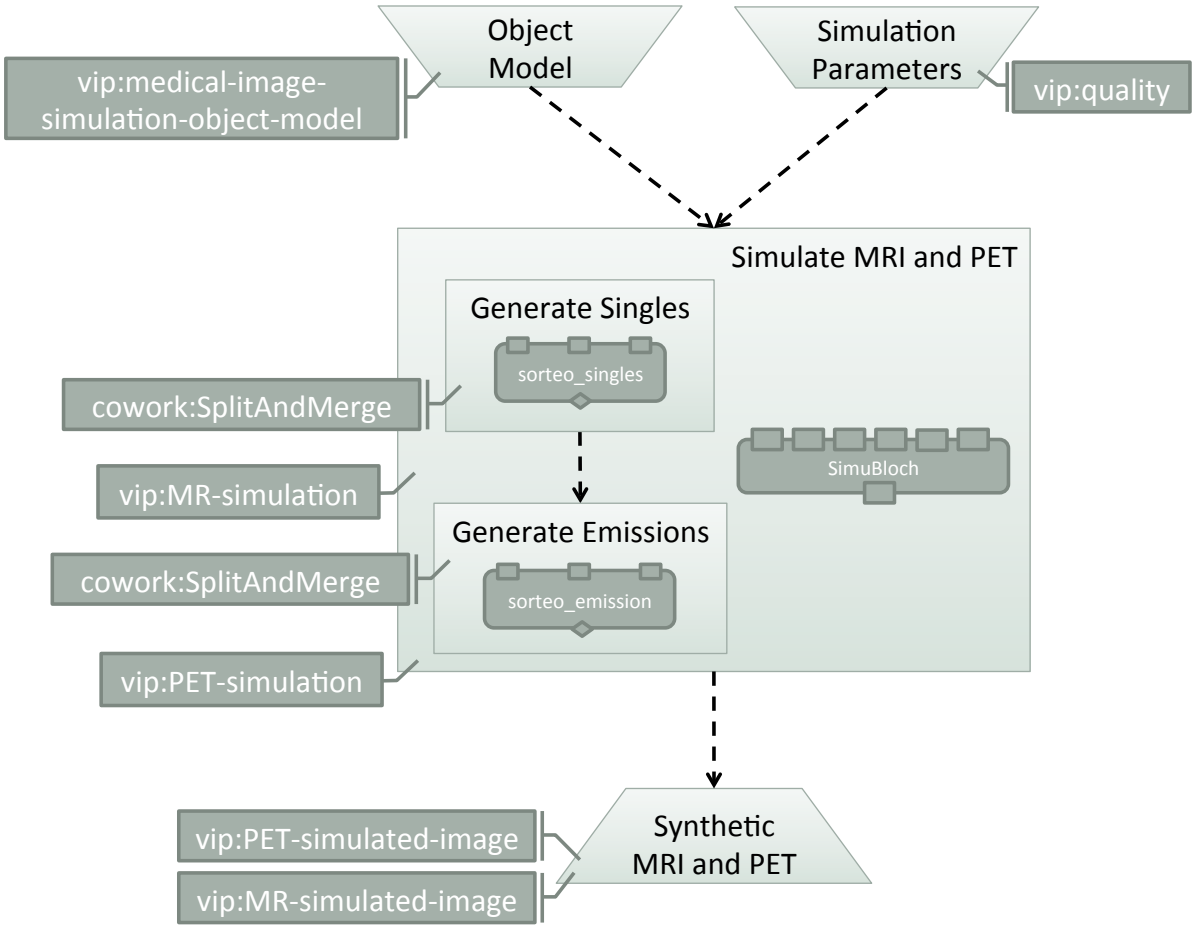


Figure 5.27: Mapping Use Case - After Weaving SimuBloch

Figure 5.27 shows the result of **Weaving** the top discovered match `SimuBloch` (shown on Figure 5.21) into the base workflow shown on the previous Figure 5.26.

Applying **Discovery** to the `Simulate MRI and PET` **Conceptual Function** obtained after **Weaving** `SimuBloch` **Fragment**, finds the following matches (Legend: `Fragment name [`*score*`]`):

1. `Split and Merge` [1.0]
2. `Node-bound Split and Merge` [1.0]
3. `PET 2 Steps` [0.25]
4. `SIMRI (complete workflow)` [0.17]
5. `SimuBloch (complete workflow)` [0.17]
6. `SORTEO (complete workflow)` [0.1]

The only **Requirement** left is `SplitAndMerge`, so the `SimuBloch` **Fragment** is no longer a match, while the score of pure `Split and Merge` **Fragments** is perfect. The only issue is which **Fragment** to pick between the link-bound and the node-bound one. In this precise case, the link-bound **Fragment** will have no effect if woven, because `Generate Singles` features no incoming **Conceptual Links** and `Generate Emissions` features no outgoing ones: there is no match for the link-bound **Pattern** in this specific workflow (shown on Figure 5.27).

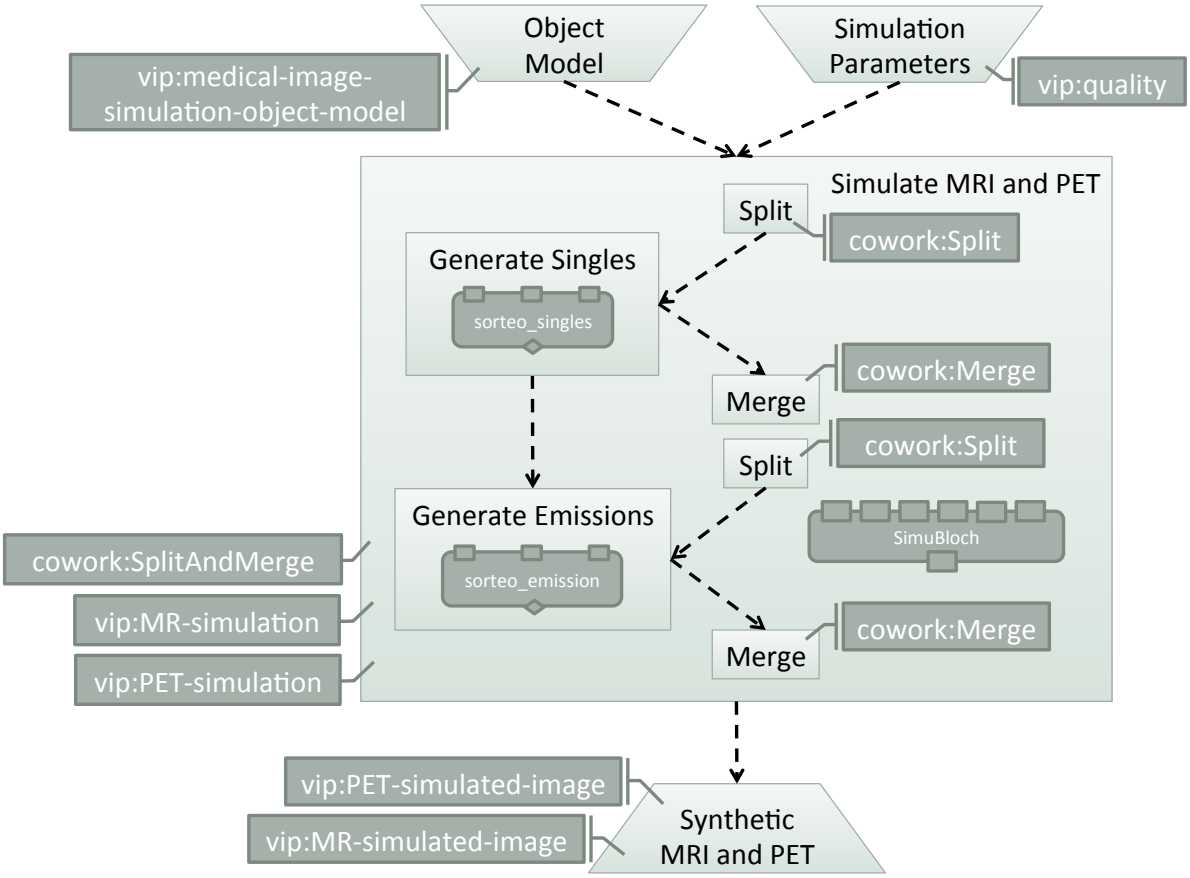


Figure 5.28: Mapping Use Case - After Weaving (node-bound) Split and Merge

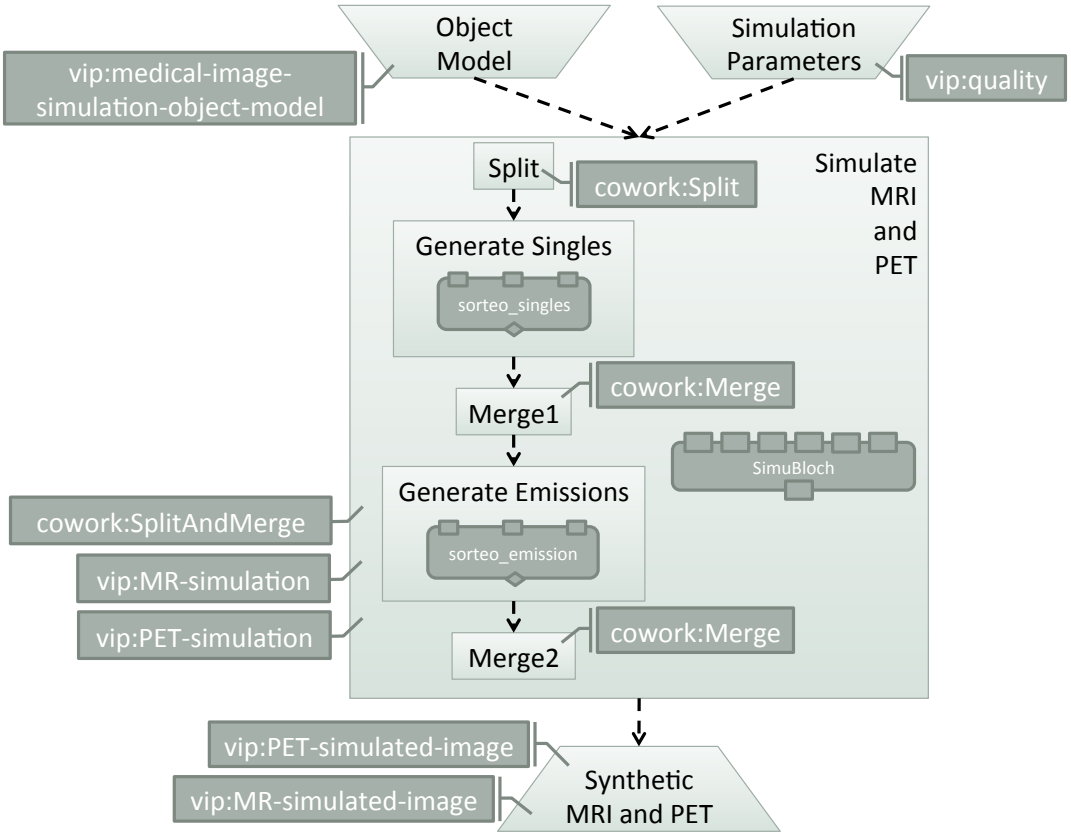


Figure 5.29: Mapping Use Case - Fixed Split and Merge Steps

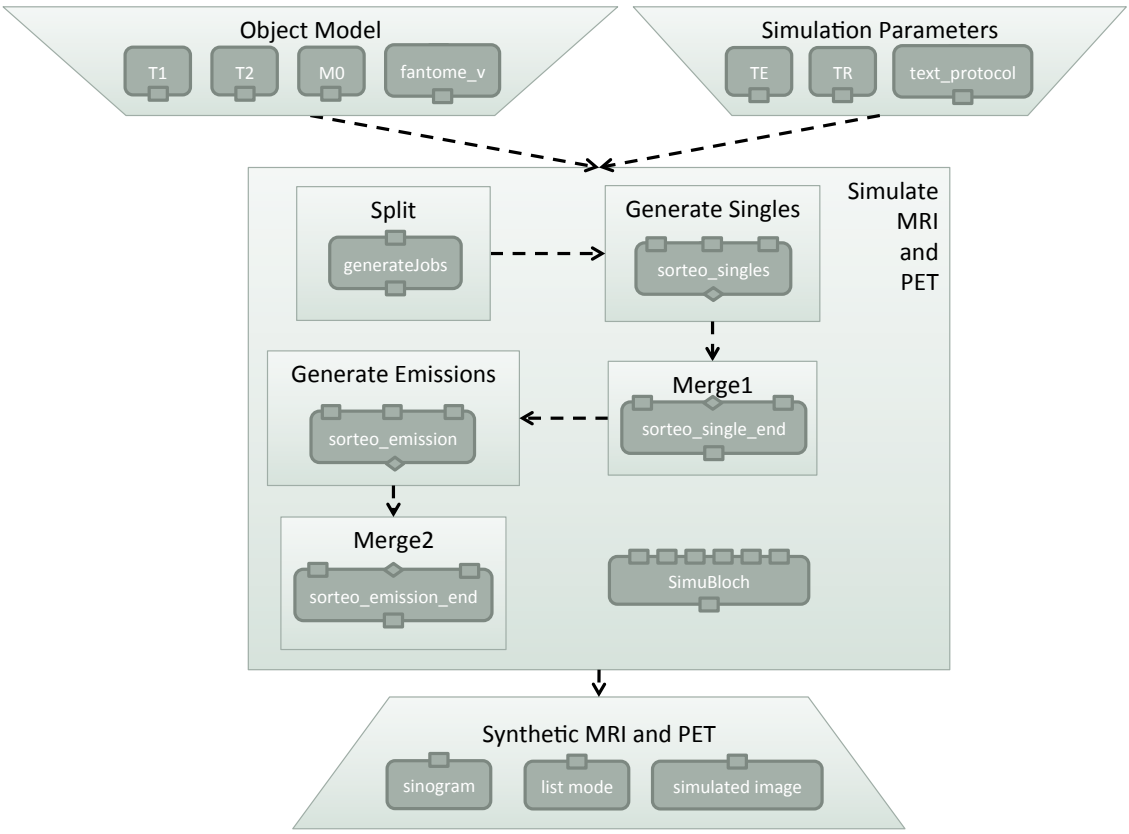


Figure 5.30: Mapped Use Case

Figure 5.28 shows the result of **Weaving** the node-bound `Split and Merge` **Fragment** (shown on Figure 5.24) into the base workflow shown on the previous Figure 5.27. That result is imperfect because it created the `Split` and `Merge` steps alongisde rather than on the path from `Generate Singles` to `Generate Emissions`. What we really want is one `Split` step at the beginning of the chain and one `Merge` step after each generation phase. Fixing that is easy enough to do manually and the result is shown on Figure 5.29.

Only **Inputs** and **Outputs** remain to map now. They can be either entered manually or woven automatically if they have been wrapped into simple **Fragments** (as described in Section 5.4.1.1). Either way, the end result, the mapped **Conceptual Workflow**, is shown on Figure 5.30.

5.4.4 Composition

In this entire section, the the following constants were used (*cf.* Section 4.4.3.2): exact matches constant $K_{EM} = 1.0$; narrower matches constant $K_{NM} = 0.5$; and broader matches constant $K_{BM} = 0.25$.

When run against the knowledge base used in the previous Section 5.4.3, the $link$ function detailed in Section 4.5.1 suggests the following links, as shown on Figure 5.31:
Legend: `source.outputPort` → `target.inputPort` [comment on quality]

- `generateJobs.out` → `sorteo_emission.jobs` [excellent]
- `generateJobs.out` → `sorteo_single_end.protocol` [broken]
- `generateJobs.out` → `sorteo_singles.jobs` [excellent]
- `generateJobs.out` → `sorteo_singles.protocol` [broken]
- `M0.out` → `simuBloch.m0` [excellent]
- `sorteo_single_end.out` → `sorteo_emission.singles` [excellent]
- `sorteo_single_end.out` → `sorteo_emission_end.singles` [excellent]
- `T1.out` → `simuBloch.t1` [excellent]
- `T2.out` → `simuBloch.t2` [excellent]
- `TE.out` → `simuBloch.te` [excellent]
- `text_protocol.out` → `generateJobs.in` [mismatched]
- `text_protocol.out` → `simuBloch.te` [broken]
- `text_protocol.out` → `simuBloch.tr` [broken]
- `text_protocol.out` → `sorteo_emission.jobs` [broken]
- `text_protocol.out` → `sorteo_single_end.protocol` [mismatched]
- `text_protocol.out` → `sorteo_singles.jobs` [broken]
- `text_protocol.out` → `sorteo_singles.protocol` [mismatched]
- `TR.out` → `simuBloch.tr` [excellent]

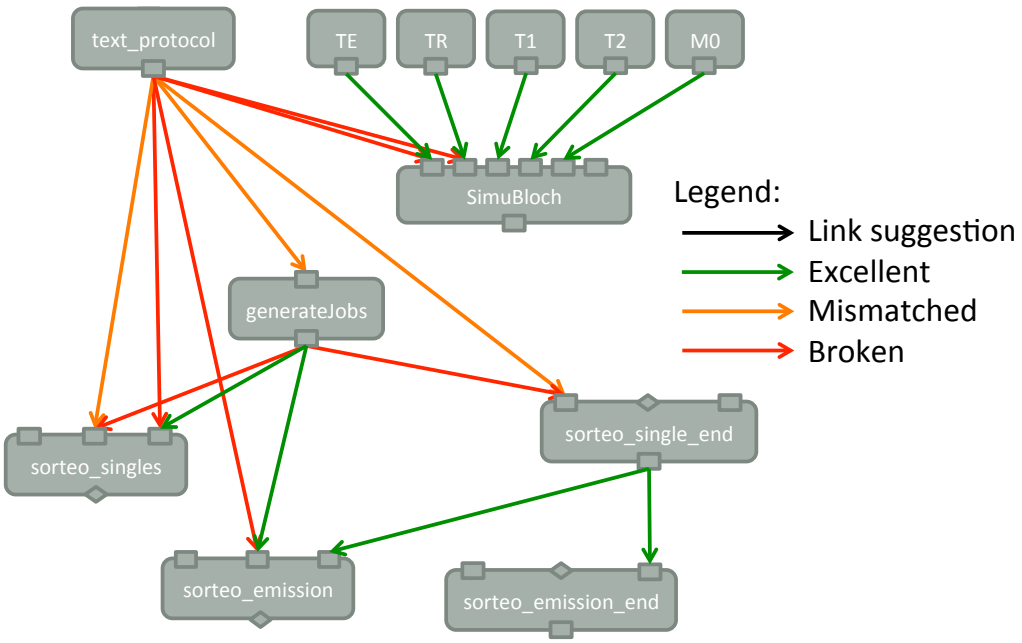


Figure 5.31: Use Case Link Suggestions

The comments on quality given here are not produced by the prototype, though mismatch detection is a planned feature. They are judgments informed by the **Conceptual Workflows** the **Activities** were fetched from. [excellent] means that the suggested **Data Link** exists as-is in the original workflow the **Activity** came from; [mismatched] means that a converter must be inserted between the source and the target for the suggested **Data Link** to work; and [broken] means that the suggestion makes little sense and is a product of the semantic gap. For instance, `generateJobs.out` and `sorteo_single_end.protocol` both bear the very generic `quality` **Annotation** but are not compatible at all.

In this specific instance, the *link* function produced 9 excellent, 3 mismatched and 6 broken suggestions, which amounts to a 66% relevancy rate, given that mismatched links can further be used to look for converters.

The same detection gap plagues the other **Composition** functions. The more generic and widely used a **Specification** is, the less relevant the suggestions based on it will be.

For instance, **Ports** `sorteo_singles.protocol` and `text_protocol.out` both bear `quality`. The *produce* function, detailed in Section 4.5.2, when applied to the former, returns the four **Activities** `copy file`, `CompileProtocol`, `generateJobs` and `parse_text_protocol` each with a perfect score. Of the four, only `CompileProtocol` fits, but **Annotations** do not reflect that. Similarly, the *consume* function, detailed in Section 4.5.3, when applied to `text_protocol.out`, returns 13 **Activities** featuring an **Input Port** annotated with `quality`, each with a perfect score, but only one is truly relevant.

Conversely, the more specific and rarely used a **Specification** is, the more relevant the suggestions will be. For instance, applying *produce* to `SimuBloch.dir` which is annotated with `directory` returns only the appropriate `appendDate` **Activity**. Even more interestingly, applying *consume* to `sorteo_emission_end.out`, which is annotated with `PET-list-mode-data`, returns the appropriate `LMF2RAWSINO` **Activity** with a perfect score, but also `sorteo_emission` and `sorteo_emission_end`, each with a score of 0.5, because they feature an **Input Port** annotated with `PET-simulated-data`, the direct supertype of `PET-list-mode-data`, and are thus broader matches.

Annotations are even more critical for converter suggestion, since it involves looking for chains of **Activities**. The more **Activities** there are for a given **Annotation**, the longer it takes to score all the alternative chains. The chain size threshold has an even bigger impact on performance.

For instance, the function *convert*, detailed in Section 4.5.4, applied on a mismatched **Data Link** from `sorteo_emission_end.out` to `sinogram.in`, bearing `PET-sinogram`, returns only the appropriate **Activity** `LMF2RAWSINO` with a perfect score.

However, on the **Data Link** from `text_protocol.out` to `generateJobs.in`, bearing the generic and well-spread `quality`, the *convert* function returns three **Activities** besides the appropriate `parse_text_protocol - copy file`, `CompileProtocol` and `generateJobs` - and all four have perfect scores.

The difference between the two cases becomes a lot starker when looking for conversion chains with two **Activities**. Indeed, from `text_protocol.out` to `generateJobs.in`, *convert* returns only one chain with `LMF2RAWSINO` used twice, with a score of 0.5, whereas from `text_protocol.out` to `generateJobs.in`, *convert* returns 40 chains with scores of either 0.5 or 0.25 depending on the number of extra **Functions** and **Concerns** of the **Activities** in the chain.

Table 5.1 indicates the average execution time observed over 10 runs on a laptop equipped with a dual-core 2.8GHz processor and 8GB RAM.

Table 5.1: Converter Suggestion Performance

Source	Target	Specification	Size	Time
`sorteo_emission_end.out`	`sinogram.in`	`PET-sinogram`	1	17 ms
`text_protocol.out`	`generateJobs.in`	`quality`	1	25 ms
`sorteo_emission_end.out`	`sinogram.in`	`PET-sinogram`	2	129 ms
`text_protocol.out`	`generateJobs.in`	`quality`	2	2341 ms

As confirmed by the execution times showed on Table 5.1, there are three factors negatively impacting the performance of converter suggestion:

- **genericity**: the more generic an **Annotation** is, the more **narrower matches** it has, thus increasing the number of potential results;
- **usage spread**: each **Activities** bearing an **Annotation** increases the number of potential results; and
- **chain size**: because all intermediary types are considered, doubling the chain size likely more than doubles the number of potential results.

Of the three factors, the chain size is by far the most limiting factor.

5.4.5 Conversion

Once the workflow is fully composed through picking the appropriate producers, consumers and converters, it becomes the **Intermediate Representation** shown on Figure 5.32. **Conversion** is then a fully automated process. Figure 5.33 shows the result of that process targeting GWENDIA. That result can be opened in MOTEUR, but has yet to be fully tested on the VIP platform.

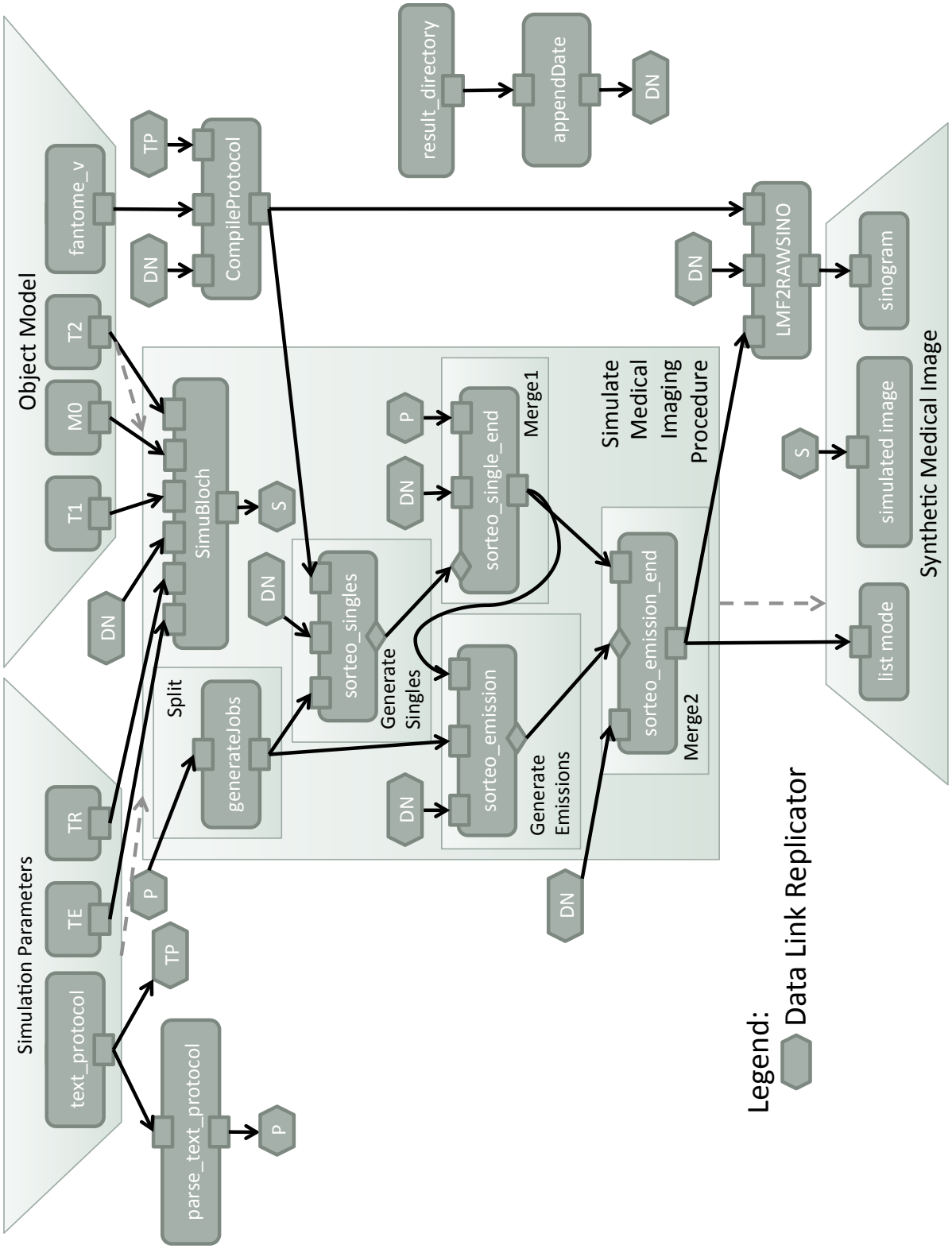


Figure 5.32: Use Case Intermediate Representation

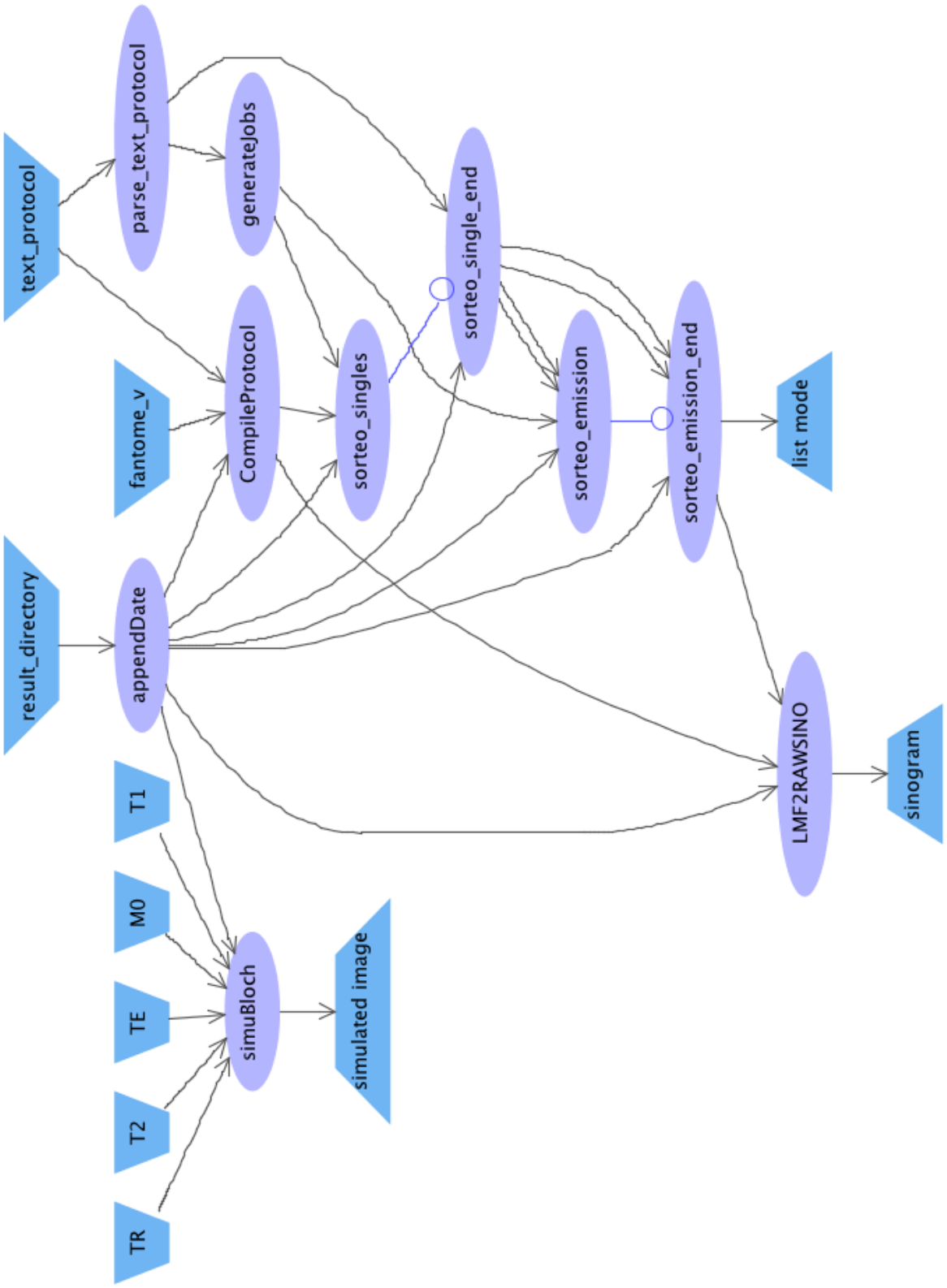


Figure 5.33: Use Case GWENDIA Conversion Result (MOTEUR screenshot)

5.4.6 Discussion

The use case, in combination with the prototype and VIP simulators, has allowed us to test the basic assumptions, the structure and algorithms of our work. It has also showed how much the quality of the computer-assistance provided by the system depends on the quality of the knowledge base. It is quite obvious that the more detailed the available semantic descriptions are, the more useful the system becomes, especially when making suggestions. Consequently, the system will no doubt suffer from the "*cold-start problem*" that is common with semantic-based systems as well as content-based approaches such as collaborative filtering, *i.e.* recommendations based on similarity between user profiles.

On the other end of the spectrum, it is also clear that the prototype as it is currently built would suffer from performance issues if used against a huge knowledge base, especially if some **Annotations** are used repeatedly in large numbers of **Fragments**. Caching of results would most probably help, but some more work on query optimization might be needed as well. Moreover, when combining the probability of quality loss with the huge performance hit, it seems conversion chains should be reserved for cases where no results are returned at all and, even then, remain optional. On the one hand, the arbitrary choices we made for the values of scoring constants (*cf.* Section 4.4.3.2) worked fine for the use case, but further study is needed to check how they might require adapatation in different contexts. On the other hand, it is clear from the use case that further heuristics are sorely needed for **Composition**, to try to discriminate between elements bearing the same **Annotations**. For instance, it should be possible to take the existing uses of an **Activity** into account when scoring it for producer, consumer or converter suggestions.

Among issues the use case highlighted is the tedium of extracting trivial **Fragments**. It should be possible to alleviate this issue by automating the extraction partially, *i.e.* detect sub-workflows fitting the template described in Section 5.4.1.1 and suggest the corresponding **Fragments** to the user. Another, maybe complementary, option would be to extend **Discovery** to **Activities** so that when a single **Activity** is adequate for the **Mapping** of a given **Conceptual Workflow**, it would not be required that it is wrapped alone in a **Fragment** for it to be found and suggested to the user.

CHAPTER SUMMARY

We built a **prototype** system to design Conceptual Workflows and implemented the various algorithms and tools of the Transformation Process. The prototype was **integrated** as a Workflow Designer into the **Virtual Imaging Platform** (VIP) and the use cases of the VIP project allowed us to ensure that the **Conceptual Workflow Model is expressive** enough to model real life simulations and that the **Transformation Process performs as well as the Knowledge Base is populated**.

CHAPTER 6

CONCLUSION

The need to automate simulations and/or run them on Distributed Computing Infrastructures (DCIs) is ever-increasing in most scientific domains. Unfortunately, most scientific workflow models mix domain goals and methods with technical aspects and non-functional concerns in such a way that scientific workflows become hard to use for scientists who are not distributed computing experts or enthusiasts. Consequently, our objective in this work was as follows:

> GOAL
> Create a new scientific workflow model to improve **accessibility**, ease **reuse** and allow **comparison**.

For us, the most obvious impediment to accessibility and reuse was the abstraction level of most scientific workflow models. We identified three distinct levels, based on existing systems and the litterature: the **Concrete** level of technical enactment, the **Abstract** level where most scientific workflow models lie and the **Conceptual** level where scientists truly conceive their simulations, before implementing them. Our first step was thus:

> METHOD (PART 1 OF 4)
> Create a new scientific workflow model at the **Conceptual Level**.

We designed the **Conceptual Workflow Model** for scientists to model their simulations at a computation-independent level, free from technicalities. Applying it to real-life examples taken from the Virtual Imaging Platform (VIP) project shows how it applies to real-life scenarios.

Though **Conceptual Workflows** could arguably be useful as documentation on their own, the end goal of designing a simulation is of course to execute it, most often on DCIs. Hence the need for a systematic way to transform **Conceptual Workflows** into abstract workflows which can be delegated to already plentiful third-party scientific workflow frameworks. Our open-world hypothesis precluded full automation, the best we could aim at was thus:

> METHOD (PART 2 OF 4)
> Develop a computer-assisted **Transformation Process** from the **Conceptual Level** to the **Abstract Level**.

The **Transformation Process** we developed is split in two steps: the **Mapping** phase relies on Semantic Web technologies and domain **Semantic Annotations** to lower the abstraction level to the **Abstract Level**; then the **Conversion** processes convert to target languages. The **Mapping** phase, like the **Conceptual Workflow Model** itself, is as independent from the chosen target language as possible, so as to not to hinder comparison. Applying the **Transformation Process** on a use case inspired by the VIP project ensured that it is working as intended, with a performance depending entirely on the content of the knowledge base.

However, a purely conceptual model would not accomodate the information required to run a simulation and would therefore not suit the needs of the **Mapping** phase, we thus had to:

METHOD (PART 3 OF 4)
Extend the scientific workflow model with elements of the **Abstract Level** to model **Intermediate Representations**.

The **Abstract Elements** we extended the **Conceptual Workflow Model** with have proven sufficient for the simulations modeled so far and, if the need arises, adding more will not be difficult.

Emphasizing domain goals and methods over technicalities is not enough to improve the legibility and ease-of-reuse of scientific workflows. Indeed, cross-cutting non-functional concerns require more specific solutions, hence the need to:

METHOD (PART 4 OF 4)
Develop technologies to weave different types of concerns into scientific workflows.

We defined **Fragments**, with **Blueprints** describing how to weave them and **Patterns** describing where; and we developed a **Weaving** algorithm which transforms **Fragments** into SPARQL Protocol and RDF Query Language (SPARQL) `CONSTRUCT` queries to leverage the semantic nature of the **Conceptual Workflow Model** and the graph transformation capabilities of SPARQL. The VIP simulators provided us with a variety of **Fragments**, more or less complex, and we could check it is functioning as intended and lets designers express cross-cutting concerns like data parallelism optimization directly in the **Conceptual Workflow Model**.

Limitations

The biggest limitation of a **Conceptual Workflow** framework is its dependency on the content of the knowledge base: if the latter contains neither **Activities** nor **Fragments** matching the **Annotations** of the **Conceptual Workflow** being mapped, then the **Transformation Process** will provide no assistance to the user. Conversely, as it is defined now, the **Transformation Process** will scale very poorly to a knowledge base filled with similarly annotated **Activities** and **Fragments**.

Weaving is a powerful tool, but its efficiency is contingent on the quality of **Fragments** and their design is not trivial. As the process stands now, **Fragments** must be designed manually by users having a deep understanding of the **Weaving** process. For the system to fit the needs of non-expert users, it would therefore be vital to develop automated **Fragment** generation features and document the **Weaving** process in a way that makes it easily understandable for prospective users.

Discovery and **Composition**, as they are defined now, rely solely on **Annotations**. Since the entire **Conceptual Workflow Model** can be browsed and queried semantically, it could be more beneficial to also take the structure of **Conceptual Workflows** into account when making suggestions.

Positionning

Using the same legend as the overview table given in Section 2.1.4, Table 6.1 highlights the positionning of our contributions against the scientific workflow frameworks we analyzed.

Table 6.1: Position of Conceptual Workflow Framework

Framework	Interface	Model	Abstraction Level			
			Annot.	Compos.	Flexib.	Indir.
COWORK	In development	Hybrid DCG	◆	◆	◆	◆

Perspectives

Aside further development of our prototype framework, there are many avenues for future research on the topic of **Conceptual Workflows**.

For instance, given that **Conceptual Workflows** let users emphasize the fundamentals of a simulation, as opposed to technical and non-functional concerns, they could potentially be used, in conjunction with provenance systems, to highlight the parts of an execution which are most relevant to the end user. Indeed, most provenance tracking systems treat all **Activities** equally, which can be confusing for anyone perusing provenance traces, since a big number of **Activities** pertain to technical or non-functional concerns and give very little information about the experience as a whole. Systems like the NeuSemStore[1] provenance tracker MOTEUR plugin [Gaignard 13] help alleviate that issue by letting users create experiment summaries focusing on the parts of the simulation that are most relevant to the users and retrieving only the provenance traces associated with those **Activities**. However, as of this writing, the set of rules used to define experiment summaries have to be created manually. **Conceptual Workflows** could conceivably be used to generate those rules automatically, emphasizing the provenance traces pertaining to **Conceptual Functions**.

Since they are not tied to a specific target language, **Conceptual Workflows** might serve as a meta-model for interoperability purposes, though that would entail designing a **Conversion** process which supports multiple languages at a time and adapting the **Mapping** process to account for it. Such a high-level meta-model would be more accessible for non-expert users than existing interoperability meta-models, which are targeted at expert scientific workflow designers, *e.g.* IWIR [Plankensteiner 11] and WS-PGRADE [Kacsuk 12] used as such on the SHIWA [Krefting 11] platform.

Also, given how tedious annotating can be, it would be interesting to investigate how **Semantic Annotations** for **Activities** (*e.g.* web services, legacy programs) could be automatically or semi-automatically deduced from their use in a **Conceptual Workflow**. Not only would that ease the enrichment of the knowledge base, it might also be useful to export those **Semantic Annotations** so they could be leveraged by other semantic systems such as the WINGS [Gil 11b] scientific workflow framework.

[1] NeuSemStore: `https://nyx.unice.fr/projects/neusemstore`

APPENDIX A
DETAILED FRAMEWORKS

Given the number and variety of scientific workflow frameworks, we could not possibly establish an exhaustive list or comparison. Our aim here is to give an overview of some of the better known systems to date, summarized in Table 2.1, shown in Section 2.1.4. Systems are presented in alphabetical order. Where two names are given, the first one is the name of the framework and the second that of the associated scientific workflow language.

A.1 ASKALON/AGWL

The ASKALON [Fahringer 07] framework is developed by the Distributed and Parallel Systems Group at the University of Innsbruck, Austria. It serves as the main interface to the Austrian Grid[1] Distributed Computing Infrastructure (DCI). Its chief objective is to lower the entry barrier to grid computing by having the enactor handle most of the technical operations pertaining to grid usage (e.g. scheduling, resource allocation, service deployment), in order to shield the end-user from that underlying complexity. Figure A.1 shows a screenshot of a simple example.

Interface The Graphical User Interface (GUI) uses the Java Web Start[2] technology and lets users create workflows either by composing them graphically or by directly editing the equivalent AGWL [Fahringer 05] document. Though compliant with eXtensible Markup Language (XML) and thus quite verbose, AGWL looks very much like an Imperative Programming Language (IPL), albeit at the higher-level of abstraction of control-driven scientific workflows.

Model The associated graphical representation is based on the diagrams defined by the Unified Modeling Language (UML) standard, especially the "*Activity Diagram*" which closely matches the typical scientific workflow Directed Cyclic Graph (DCG) model. At first glance, ASKALON might seem hybrid, since AGWL provides plenty of control constructs (e.g. "*forEach*" loops, "*switch*" conditionals) but also explicit "*data flows*". However, control constructs are given such precedence over so-called "*data flows*" in the language that:

[1] Austrian Grid: `http://www.austriangrid.at/`
[2] Java Web Start: `http://bit.ly/javawebstart`

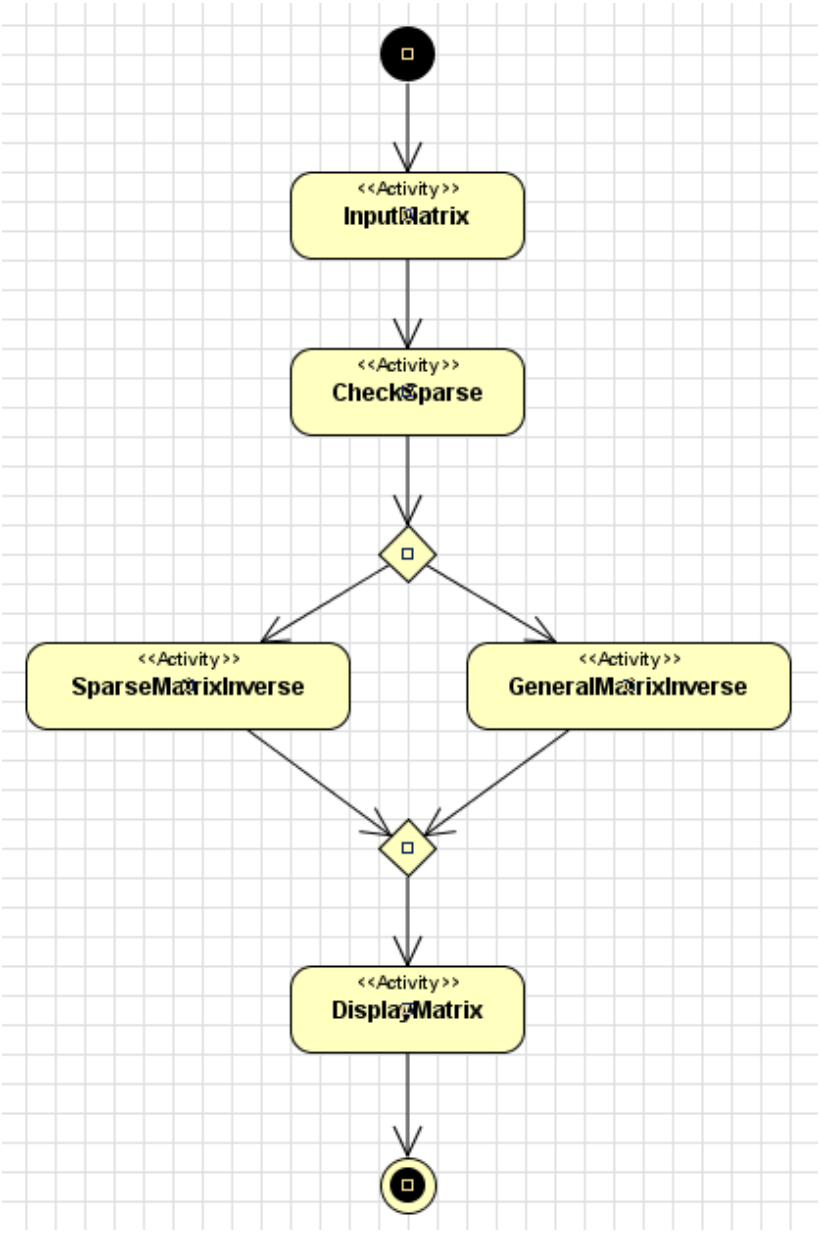


Figure A.1: Sample ASKALON Workflow

> *“ For each activity in AGWL it must be guaranteed that whenever the control flow reaches the activity, all the data-in ports of the activity have been assigned to well-defined values (valid data packages). When the control flow leaves, all its data-out ports must be well-defined as well.*
>
> [FAHRINGER 05]

In other words, the data flow itself must be handled explicitly through control constructs, which is why we argue that while there is a way to explicitly route data from one activity to another, the flow of execution is not determined by data and ASKALON is therefore control-driven.

Abstraction Level Though ASKALON shields the end-user from many complex aspects of grid usage, it sits firmly at the **Abstract Level**: technical aspects of the application itself, independent from the DCI it runs on, are very much left for the user to handle. There are however ongoing efforts to use domain knowledge in order to assist workflow design [Qin 08] as well as the opposite, *i.e.* semi-automatically create domain ontologies based on workflows [Malik 12].

A.2 Galaxy

The Galaxy [Goecks 10] framework is developed mainly by the Nekrutenko lab, at the University of Penn State, USA and the Taylor lab, at Emory University, USA. It targets bioinformaticians and puts a lot of emphasis on accessibility and reproducibility. Figure A.2 shows a screenshot of a highly-rated workflow published publicly[3].

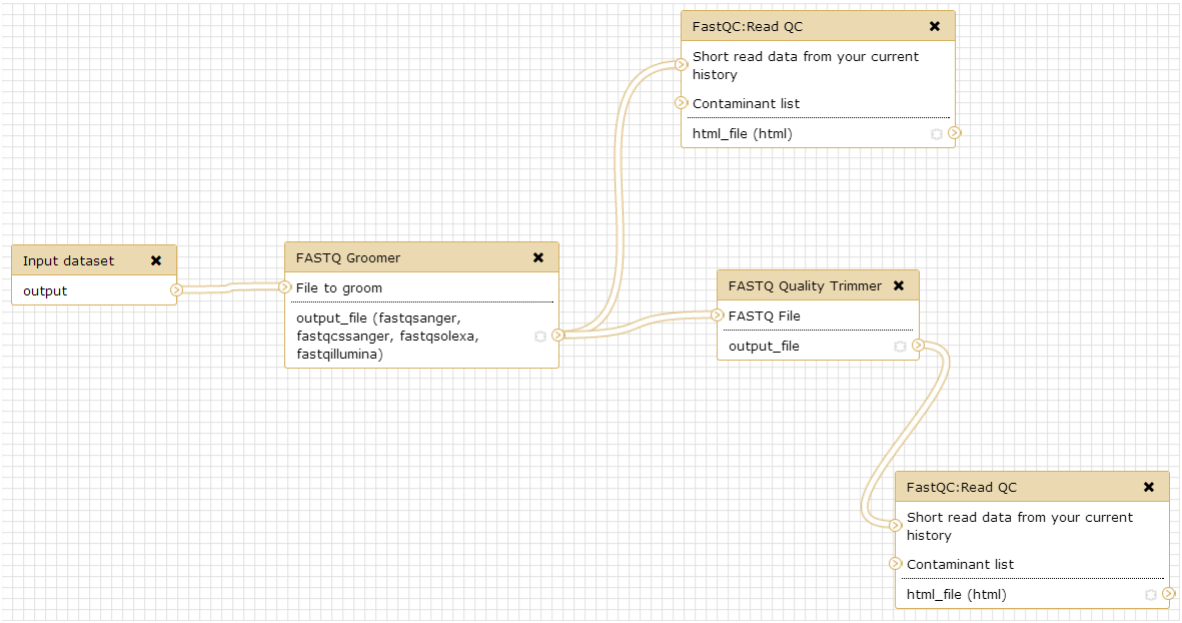


Figure A.2: Sample Galaxy Workflow

Interface Galaxy is an open-source web-based platform that can be deployed publicly or locally. The interface includes portal-like access to the "*Tools*" (*i.e.* activities) and a graphical workflow editor, as can be seen on the free public server[4].

Model The core model of Galaxy is quite obviously a DCG, as highlighted by Figure A.3. It is also quite apparent from a user perspective that Galaxy has a purely data-driven model: workflows are built by drag-and-dropping "*Tools*" from the list of available ones and creating data links between outputs and inputs.

Abstraction Level Considerable effort has clearly been devoted to make Galaxy accessible to the genomics community. Still, technicalities, *e.g.* file formats, and domain considerations, *e.g.* alignment, are handled indiscriminately. Galaxy thus lies at the **Abstract Level**.

[3]Sample Galaxy Workflow: `http://bit.ly/galaxyexample`
[4]Galaxy free public server: `https://main.g2.bx.psu.edu/`

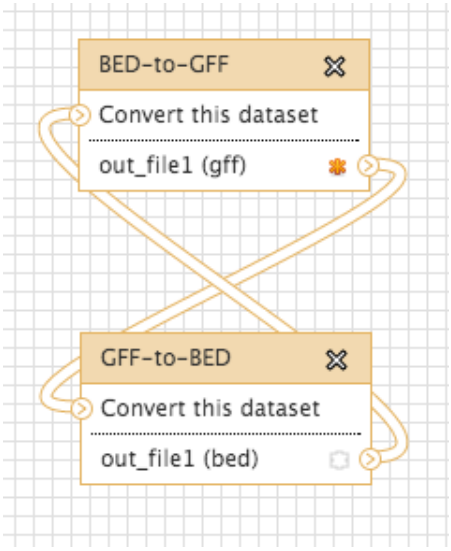


Figure A.3: Galaxy Dummy Cycle Example

A.3 GWES/GWorkflowDL

The GWES [Neubauer 05] framework is developed by the Fraunhofer Institute for Open Communication Systems, Germany. It was originally implemented for the Fraunhofer Resource Grid[5], but has since been redesigned and used in a great variety of grid computing projects. It is distributed as open-source software under the Fraunhofer FIRST license, which allows scientific and educational uses. Figure A.4 shows an example published by Andreas Hoheisel on myExperiment[6]).

Interface In order to use GWES, both a server and client(s) must be deployed as web applications. GWorkflowDL [Alt 05] workflows can be edited manually or composed via the GWES GUI. They can then be executed and monitored through either the GUI, the web services, the com-mand-line tool or customized portals.

Model The core GWES model is based on petri nets. The graph vertices thus alternate between places (*i.e.* data tokens) and transitions (*i.e.* operations). Data tokens can be flagged as "*control*", which explicitly makes them dummy data tokens. As explained in Section 2.1.2.3, the use of dummy data tokens is the most straightforward way to emulate control constructs in a data-driven model.

Abstraction Level Though, at first glance, GWES seems to firmly sit at the **Abstract Level**, a closer look at the specification of GWorkflowDL reveals a more complex picture. In order to tackle the inherent instability and dynamicity of grids and other DCIs, GWorkflowDL allows multiple levels of specification for workflow nodes: from the **Conceptual Level** with nothing more than a **Semantic Annotation** to the **Concrete Level** with a specific hardware/software instance, through the **Abstract Level** with a list of candidates.

[5]Fraunhofer Resource Grid: `http://www.fhrg.fraunhofer.de/`

[6]GWES workflow pattern by Andreas Hoheisel: `http://www.myexperiment.org/workflows/584.html`

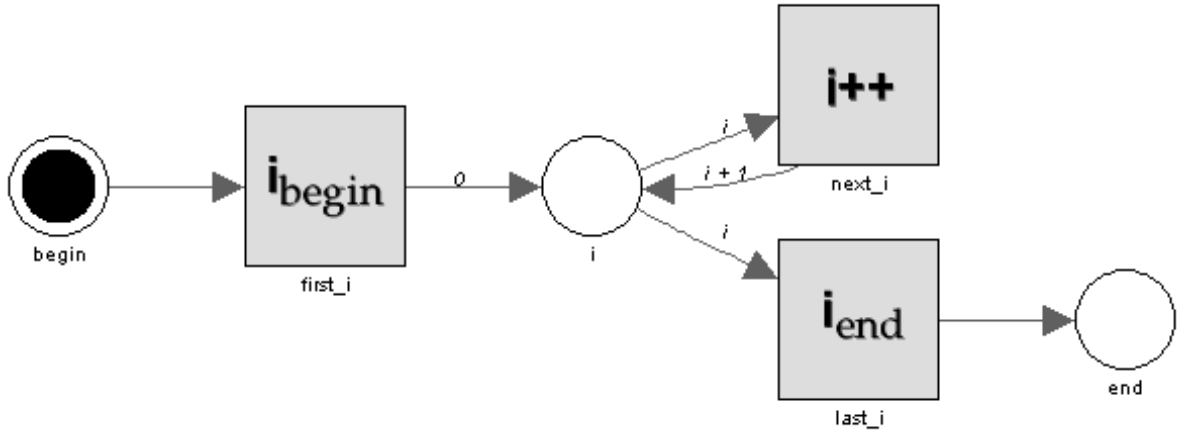


Figure A.4: Sample GWES Workflow

A.4 Java CoG Kit/Karajan

The Java CoG Kit [von Laszewski 01] is an integral part of the Globus Toolkit[7], an open-source grid building software toolkit, and is mainly developed at the Argonne National Laboratory, USA. It is as much a scientific workflow framework as a middleware upon which to build portals, such as the Open Grid Computing Environments (OGCE) portals [Alameda 07].

Interface The scientific workflow framework provides four interfaces to its XML-based scientific workflow language, Karajan [von Laszewski 07]: a GUI, a command-line tool, a web service and a scripting interface in the shape of a concise (no longer XML-compliant) version of the language called "*K*".

Model Java CoG Kit was originally meant as a client-side grid middleware: an interface between Java applications and Globus grids [von Laszewski 01]. It became a scientific workflow framework with the introduction of GridAnt[8], a workflow engine based on the Java build system Apache Ant[9], inheriting its structure of data-dependent targets and thus purely data-driven with one caveat: parallelism is not inferred from dependencies and must be explicitly specified. Deemed too limited, it was replaced by Karajan [von Laszewski 07]: a scalable grid workflow enactor and a purely control-driven model that boasts a huge number of control constructs and extensions.

Abstraction Level To the best of our knowledge, Karajan does not fulfill any of the criteria we chose to focus on for the abstraction level. However there are works that use Karajan as an enactor and use Semantic Web technologies to assist composition [Cannataro 07] or introduce indirection via a "*Resource Selector*" [Silva 08].

[7]Globus Toolkit: `http://www.globus.org/toolkit/`
[8]GridAnt: `http://www.globus.org/cog/projects/gridant/`
[9]Apache Ant: `http://ant.apache.org/`

A.5 Kepler/MoML

The Kepler [Ludäscher 06] framework is mainly developed by the Kepler/CORE team at the University of California (Davis, Santa Barbara and San Diego), USA. As a widely-known open-source project, it gets contributions from various projects and individuals outside its home institution. It is built upon Ptolemy II[10], an actor-oriented software framework developed at the University of California (Berkeley), USA. Figure A.5 shows an example taken from training session I, section 2.3.2 of a tutorial by Michal Owsiak[11].

Interface Unlike AGWL [Fahringer 05], GWENDIA [Montagnat 09] or SCUFL [Oinn 04], users are not invited to use Kepler's underlying language MoML [Lee 00] directly. To the best of our knowledge, the latest specification made publicly available dates back to 2000. The GUI is by far the preferred interface to edit, execute and monitor Kepler workflows. There is also a command-line tool and an interest group[12] dedicated to developing web interfaces for Kepler.

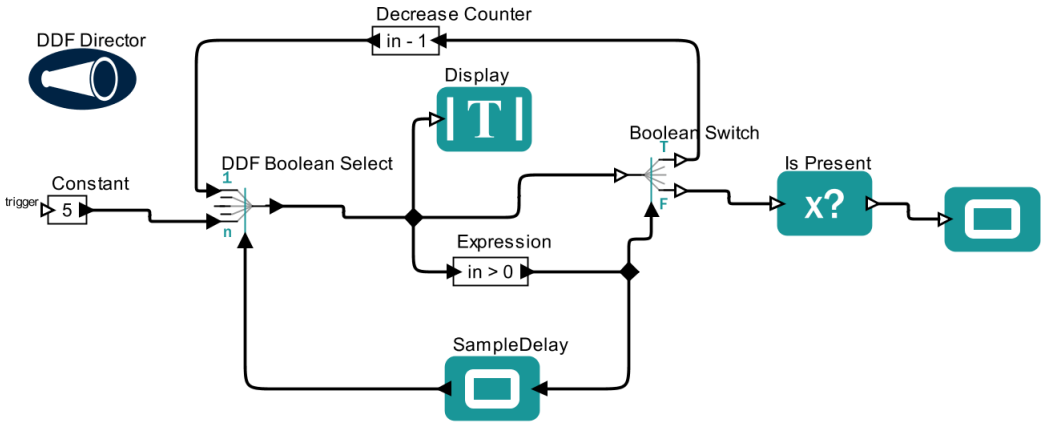


Figure A.5: Sample Kepler Workflow

Model The system aims to be a jack-of-all-trades among scientific workflow frameworks, by bringing grid computing and web services to the desktop and catering to users ranging from grid engineers to analytical scientists. Such versatility in goals is reflected in what is "*arguably the most unique feature of Kepler [which] comes from the underlying Ptolemy II system*": actor-oriented modeling [Ludäscher 06]. The core Kepler model is a data-driven DCG whose nodes represent "*actors*". At first glance, it may seem like a straightforward data-driven model, but a special actor called the "*director*" is attached to the workflow and specifies the precise semantics of the edges regarding execution flow: the Model of Computation (MoC). Among directors packaged with Kepler, the one closest to data-driven models as presented in Section 2.1.2.3 - inherently asynchronous data-flow - is "*Process Network*". If it were the only director available, the model would easily be classified as data-driven. There are however many other directors, such as "*Discrete Events*", which fires actors up according to a timeline, providing extremely fine-grained and specialized control over execution that would be hard to implement with even the most expressive control-driven model. As a result, the Kepler model is undoubtedly hybrid, but of its very own blend.

[10]Ptolemy II: `http://ptolemy.eecs.berkeley.edu/ptolemyII/`
[11]Kepler training by Michal Owsiak: `http://bit.ly/keplertutorial`
[12]Kepler Web User Interface Interest Group: `http://bit.ly/keplerwebui`

Abstraction Level Though it is one of the most accessible scientific workflow frameworks, as it stands now Kepler sits firmly at the **Abstract Level**. There are however many ongoing works (*e.g.* [Altintas 05, Bowers 06, Chin 11]) to leverage domain ontologies to further shield the user from technical details and thus elevate the abstraction level of Kepler workflows.

A.6 KNIME

The KoNstanz Information MiNer, KNIME [Berthold 07], is developed at the University of Konstanz, Germany. It is an open-source scientific workflow framework dedicated to data-mining. The eponymous company provides commercial licenses, support and extensions. Figure A.6 shows an example provided with the framework.

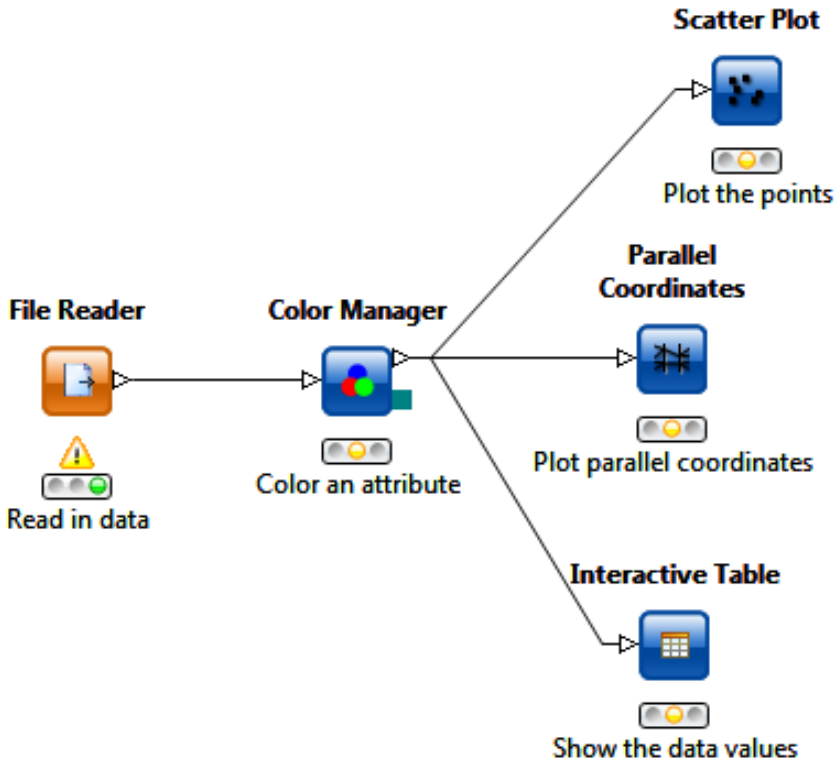


Figure A.6: Sample KNIME workflow

Interface The KNIME GUI is based on Eclipse[13] and is quite obviously tailored toward workflow monitoring, if only because of the traffic lights displayed under each activity. The commercial product "*KNIME Server*", built on top of the open-source "*KNIME Desktop*", reportedly provides various interfaces, including web portals and Application Programming Interfaces (APIs).

Model The core model of KNIME is a Directed Acyclic Graph (DAG) whose edges represent data flow. There is however an extensive number of control constructs provided as standard nodes and order constraints can be created by binding "*Flow Variable Ports*". The model is thus a data-based hybrid model.

[13]Eclipse: `http://www.eclipse.org/`

Abstraction Level While ample efforts have obviously been spent on making the KNIME interface as easy to use as possible, technical details and non-functional concerns are handled in exactly the same way as domain concerns. The KNIME model lies quite clearly at the **Abstract Level**. As of this writing, we do not know of any initiative to improve Separation of Concerns (SoC) in KNIME or to use ontologies in order to assist workflow composition.

A.7 MOTEUR/GWENDIA

MOTEUR [Glatard 08] is developed by the MODALIS team at the I3S Laboratory, France. It aims at the best trade-off between expressiveness/user-friendliness and execution performance on grids. The first version of MOTEUR used SCUFL [Oinn 04], the language of Taverna [Missier 10a], but enacted it in a fully asynchronous way on grids. However, the requirements of a remote grid workflow enactor differ sensibly from those of a desktop web services workbench such as Taverna and they led to the specification of a dedicated language called GWENDIA. Figure A.7 shows a simple example.

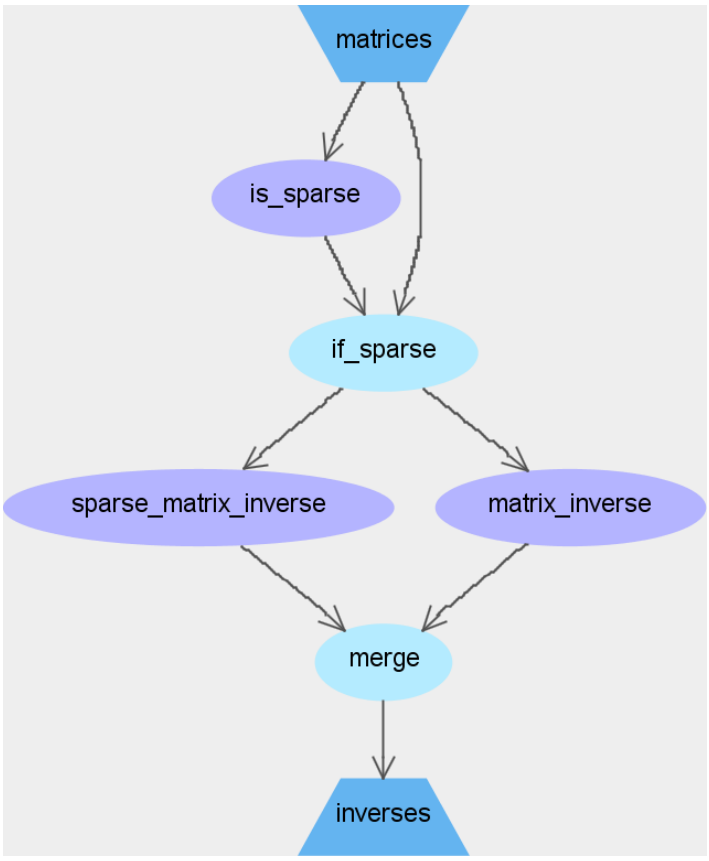


Figure A.7: Sample MOTEUR Workflow

Interface A scripting interface called "*gscript*" was recently developed [Maheshwari 10] to complement the GUI based on Graphviz[14]. There is also a web service developed in collaboration with the eBioScience group at the Academic Medical Center of Amsterdam, Netherlands and the Creatis Laboratory, France.

[14]Graphviz: `http://www.graphviz.org/`

Model GWENDIA "*targets the coherent integration of (1) a data-driven approach; (2) arrays manipulation; (3) control structures; and (4) maximum asynchronous execution capabilities*" [Montagnat 09]. With the specification of GWENDIA, the model thus evolved from pure data-driven DAG to hybrid DCG.

Abstraction Level Like most scientific workflow frameworks, MOTEUR lies at the **Abstract Level**. Though it is not restrained to MOTEUR as a target language, the present work can be seen as ongoing work to elevate MOTEUR to the **Conceptual Level**.

A.8 Pegasus/DAX

The Pegasus [Deelman 05] framework is developed mainly at the University of Southern California, USA. It is focused on enactment over a variety of DCIs and many more user-friendly systems are built upon it, notably WINGS [Gil 11b].

Interface There is no dedicated GUI for Pegasus, though projects like WINGS have been built on top of it and could be seen as very sophisticated GUIs. Though it is possible to create DAX workflows manually or generate them directly in XML, the manual clearly recommends using the APIs instead.

Model The core Pegasus model is a DAG based on petri nets, the vertices alternating between jobs (*i.e.* activities) and files (*i.e.* data). It is purely data-driven.

Abstraction Level Pegasus is not really meant for direct use by scientists of the workflow domains: it is an underlying technology upon which many other projects and portals are built. Even though the manual describes DAX workflows as "*Abstract Workflows*" and minimal flexibility in job definitions, Pegasus is a typical example of the **Concrete Level**.

A.9 SHIWA/IWIR

The objective of the SHIWA [Krefting 11] platform is scientific workflow framework interoperability, including ASKALON [Fahringer 07], MOTEUR [Glatard 08], Triana [Taylor 07b] and WS-PGRADE [Kacsuk 12]. At the execution level, "*coarse-grained interoperability*" is achieved by creating an instance of a compatible enactor to execute a non-native sub-workflow. At the workflow level, "*fine-grained interoperability*" requires transformation between scientific workflow models so that an enactor can execute native translations of non-native sub-workflows. In order to achieve that latter level of interoperability, the approach taken by the SHIWA project is that of a pivot language all systems must be able to read from and translate to: IWIR [Plankensteiner 11].

Interface The SHIWA platform is accessed through the dedicated SHIWA Portal[15]. Though the portal provides a Java applet to edit meta-workflows (for coarse-grained interoperability), there is no direct interface to create IWIR workflows as of yet. Users are invited to export their native workflows to IWIR via tools developed in the associated framework.

[15] SHIWA Portal: `http://bit.ly/shiwaplatform`

Model At first glance, IWIR seems to be a control-based hybrid, because it features extensive control constructs as well as so-called "*data links*". But those data links merely mean that input data taken by a given task comes from a given source (either an input of the block the task is in, or an output of another preceding task). In data-driven models, data links also imply that a task is fired when it receives input on all its input ports, but the data flow that can be specified in IWIR does not impact workflow execution, therefore it is a control-driven model.

Abstraction Level As of yet, IWIR is not meant to be used directly. It could be thought of as an assembly language that some scientific workflow frameworks can translate to and/or execute. By design, IWIR sits more on the **Concrete Level**, though it can reproduce the level of indirection in task definition that is commonly available in abstract scientific workflows languages.

A.10 Swift

Swift [Zhao 07] is developed mainly at the Argonne National Laboratory, USA. It was originally inspired by the now seemingly defunct Chimera and GriPhyN projects, which are closely related to Pegasus [Deelman 05]. It relies on Karajan [von Laszewski 07] to enact workflows, but it puts a lot more emphasis on implicit parallelism. Swift focuses on scalability, targets grid and cloud computing and is reportedly used in very varied application domains.

Interface Swift is a scripting language and, as such, is a lot more concise than textual representations of other scientific workflow frameworks.

Model Because there is no graphical representation associated with Swift, the underlying model is quite different from that of most other scientific workflow frameworks. The language itself is strongly typed and functional, but also features many control constructs.

Abstraction Level Though it features no graphical representation and thus may seem a lot more technical than most other scientific workflow models, the real reason why it lies at the **Concrete Level** is because every "*app*" is specified completely.

A.11 Taverna/SCUFL

Taverna [Missier 10a] is developed by the myGrid[16] team which is mainly based at the universities of Manchester, Southampton and Oxford, UK. It is tightly integrated with the myExperiment[17] social website dedicated to sharing scientific workflows and associated documents. Its user base is composed primarily of bio-informaticians, it targets essentially web services and it focuses on ease-of-use. Figure A.8 shows a simple example.

Interface By far, the main Taverna interface is a GUI called "*Taverna Workbench*". It allows users to create/edit/run workflows as well as visualize current and previous result data. It is quite seamlessly integrated with the myExperiment platform. There is also a command-line tool and a server application to execute workflows remotely.

[16] myGrid: `http://www.mygrid.org.uk`
[17] myExperiment: `http://www.myexperiment.org/`

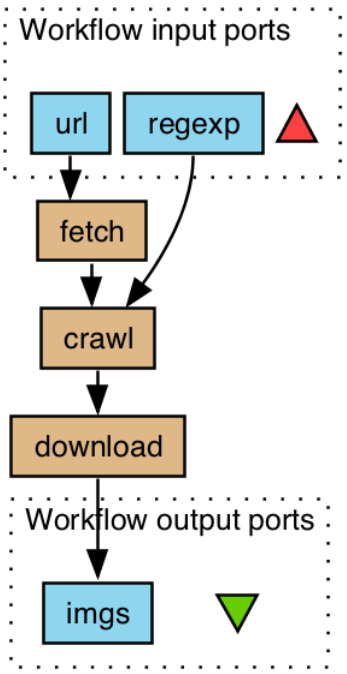


Figure A.8: Sample Taverna Workflow

Model Both Taverna and its underlying workflow language SCUFL [Oinn 04] have undergone a complete redesign around 2008 [Sroka 09]. Another significant redesign is currently ongoing, as of this writing, as shown on the Taverna Roadmap[18]. `Taverna 1` uses a dedicated language called SCUFL which is a purely data-driven DAG. `Taverna 2` dropped SCUFL in favor of a serialization format called `t2flow` and hybridized by adding control links. According to the website, `Taverna 3` will use a new version of SCUFL called `SCUFL2`, which might become even more hybrid.

Abstraction Level As it is, Taverna stands clearly at the **Abstract Level**, but there are ongoing works to elevate its abstraction level, notably to (i) use semantic annotations to improve service discovery [Wolstencroft 07, Withers 10], (ii) use metadata to guide composition [Bechhofer 10] and (iii) elevate the abstraction level of service definitions [Missier 10b].

A.12 Triana

Triana [Taylor 07b] is developed at Cardiff University, UK. It is an open-source (shared on GitHub[19]) scientific workflow framework dedicated to data analysis pipelines and it is completely integrated with the GridLab[20] grid application toolkit and testbed. Triana is compatible with many types of DCIs: the usual web services and grids, but also peer-to-peer environments. It was originally built as a tool to analyze gravitational wave data and was extended first to other types of data analyses, then to distributed primitives such as web services and grid jobs. Figure A.9 shows a simple example.

[18]Taverna Roadmap: `http://www.taverna.org.uk/introduction/roadmap/`
[19]Triana GitHub: `https://github.com/CSCSI/Triana`
[20]GridLab: `http://www.gridlab.org/`

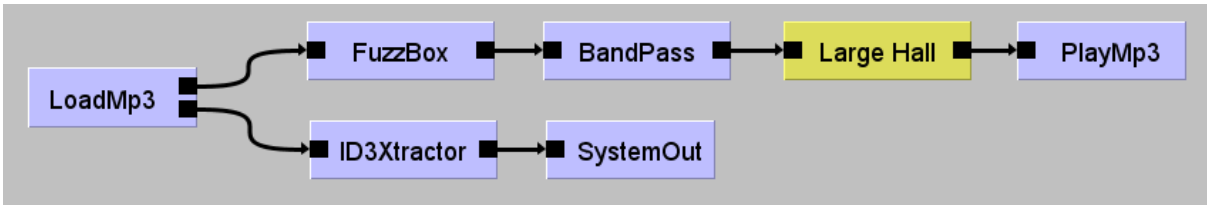


Figure A.9: Sample Triana Workflow

Interface To the best of our knowledge, the only interface to Triana is its GUI. It is however possible to package Triana workflows as self-contained applications executable from a command-line.

Model The underlying model is data-driven by construction and is even described by its creators as "*a data-flow system*" [Taylor 07b]. The core model is a data-driven DCG whose nodes are "*components*" (atomic algorithms, processes, services, etc) and whose edges represent pure data flow. Control constructs such as iterations and conditionals can only be handled through dedicated components, but it is important to note that they are much less needed in data analysis pipelines (which are the primary target of the framework) than in other types of scientific workflows.

Abstraction Level Like most scientific workflow frameworks, Triana clearly lies at the **Abstract Level**. As of this writing, we do not know of any effort to improve SoC in Triana or to use ontologies to assist the composition of Triana workflows.

A.13 VisTrails

The VisTrails [Callahan 05] framework was initially developed at the University of Utah, USA and is now mainly developed at the University of New York, USA. It focuses on provenance and takes the notion further than most: VisTrails is built to record provenance data about workflow design and not just workflow executions. Plugins are distributed by VisTrails Inc.[21] to apply their provenance technology to other things besides scientific workflows, *e.g.* 3D animation software Autodesk Maya[22]. Figure A.10 shows the `plot` example distributed with the application.

Interface The VisTrails software is a stand-alone BSD-licensed open-source application prebuilt for Windows and Mac. The GUI integrates many interfaces, including one to design workflows, one to examine a workflow's design history and one to query provenance.

Model The model underlying VisTrails is quite clearly data-driven: the user builds workflows by dropping "*modules*" onto the main window and creating "*connections*" between strongly typed output and input ports. Though the graph-based model of VisTrails is sometimes described as "*acyclic*" [Gaspar 10], it is actually a DCG, as highlighted by Figure A.11.

[21] VisTrails Inc.: `http://www.vistrails.com/`
[22] Autodesk Maya: `http://www.autodesk.com/products/autodesk-maya`

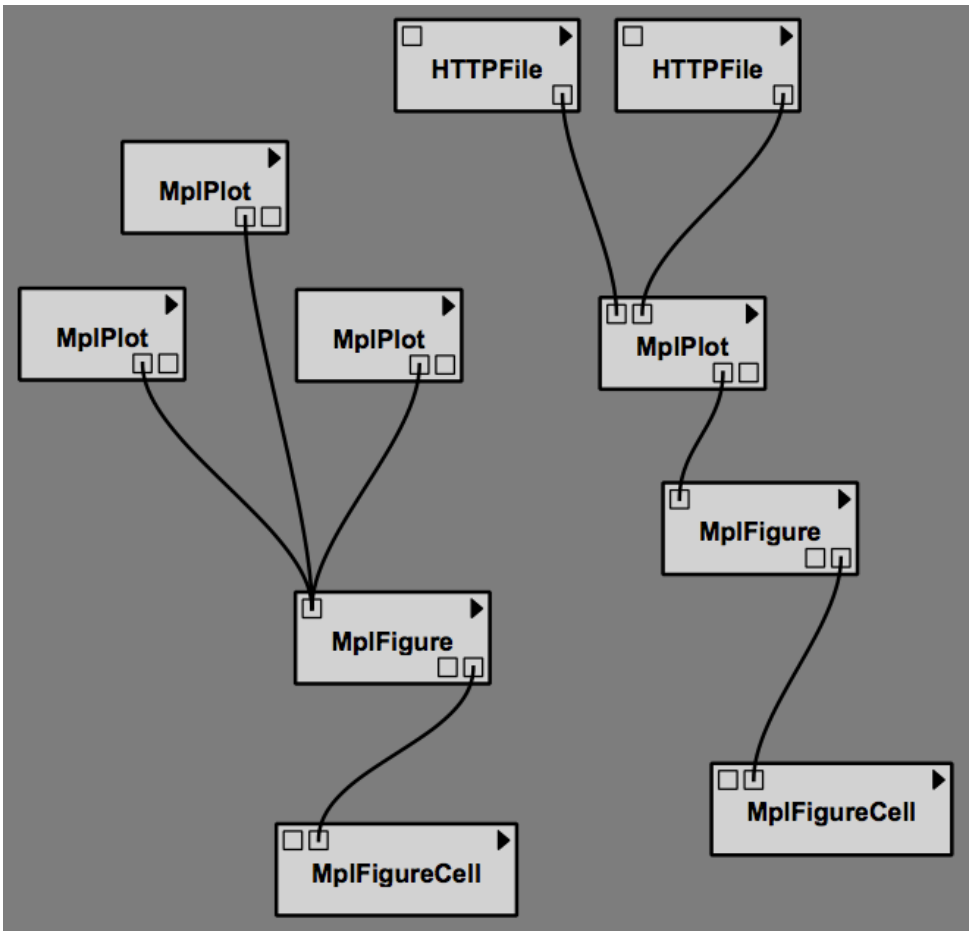


Figure A.10: Sample VisTrails Workflow

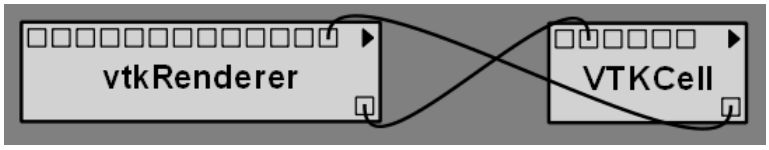


Figure A.11: VisTrails Dummy Cycle Example

Abstraction Level VisTrails itself clearly lies at the **Abstract Level**: "*modules*" are executable artifacts and can bear any number of "*key/value*" annotations. [Gaspar 10] introduces an ontology to model VisTrails workflows and ease their composition, but that effort seems independent from the main project.

A.14 WINGS

WINGS [Gil 11b] is an open-source (shared on GitHub[23]) scientific workflow framework developed mainly at the University of Southern California, USA. It is a semantic framework built on top of Pegasus [Deelman 05] to serve as a high-level accessible and domain-oriented interface to scientific workflows. Figure A.12 shows the `Words abs` template workflow accessible on the basic WINGS public server[24].

[23]WINGS GitHub: `https://github.com/varunratnakar/wings`
[24]WINGS Sandbox: `http://www.wings-workflows.org/sandbox/`

Interface WINGS workflows are built and executed on a server. There are two public servers (one with more advanced features) available for users to try WINGS and anyone can deploy their own server. To the best of our knowledge, there is no other interface to WINGS workflows, which makes sense given that it targets scientists rather than scientific workflow experts, who would probably rather use Pegasus directly.

Model WINGS inherits its core model from Pegasus and is thus also a data-driven DAG.

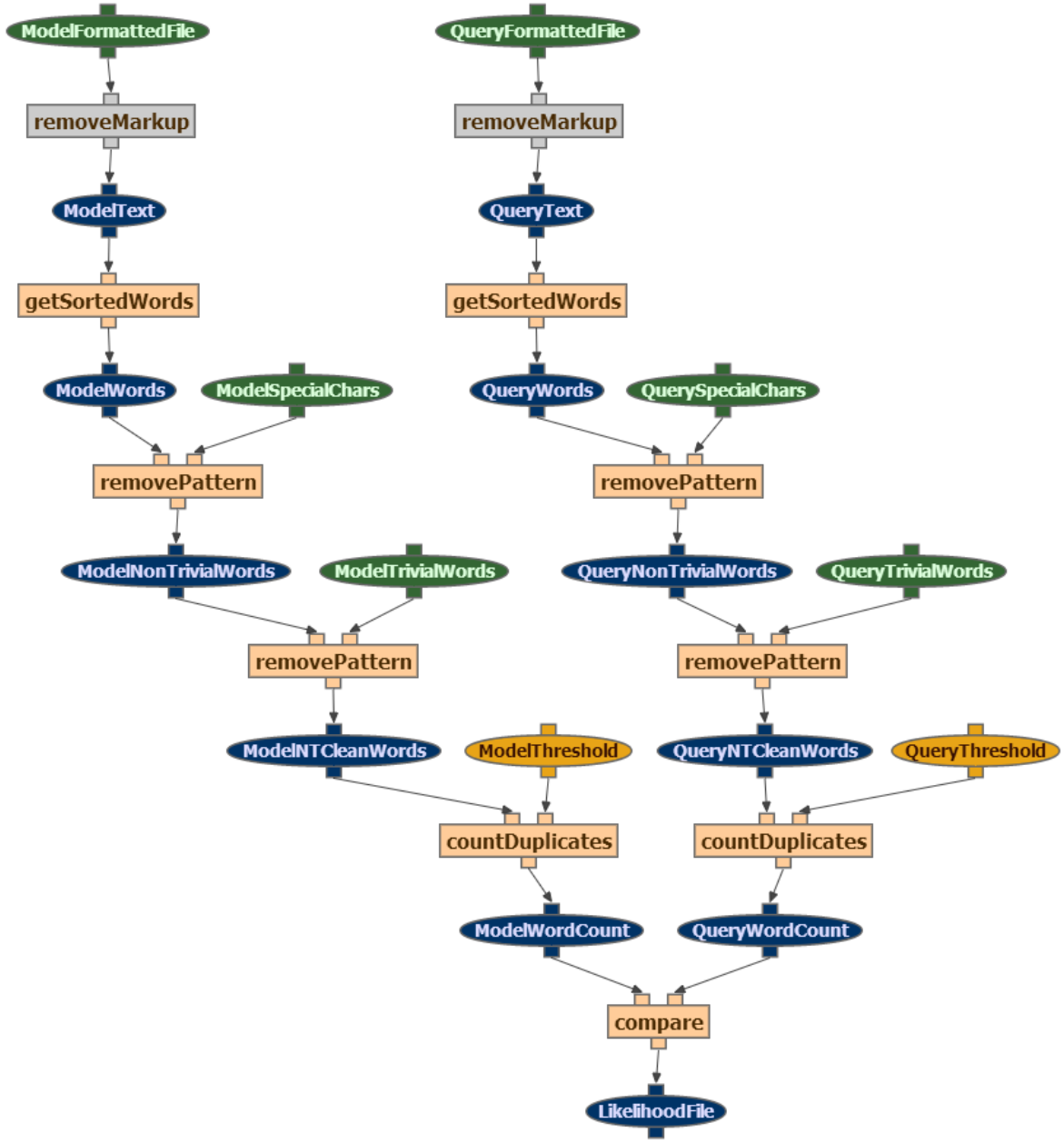


Figure A.12: Sample WINGS Workflow

Abstraction Level WINGS was built from the start as a semantic framework meant to focus on user domains. Most of the works around WINGS deal with leveraging semantic data and ontologies to ease and assist the design and sharing of scientific workflows.

Notably: [Gil 11a] describes a framework built on top of WINGS to automatically transform user queries into scientific workflows; [Garijo 11] describes an approach to publish "*abstract workflows*" (*i.e.* workflow templates with undetermined Activities) and "*executable workflows*" (*i.e.* what we call abstract workflows) as Open Linked Data through an extension of the Open Provenance Model[25] (OPM); [Hauder 11] focuses the framework on the state of the art in data mining pipelines and obtains great results on automated composition and increased accessibility; and [Garijo 13] mines provenance data to detect "*abstract templates*" and thus elevate the abstraction level of WINGS workflows automatically. There is no doubt that WINGS is the most well-known and furthest developed **Conceptual Level** scientific workflow framework to date.

A.15 WS-PGRADE

WS-PGRADE [Kacsuk 12] is an open-source - under APACHE license - scientific workflow framework and also the GUI/portal layer of the grid and cloud User Support Environment (gUSE)[26] DCI gateway. It is mainly developed at the Laboratory of Parallel and Distributed Systems, MTA-SZTAKI, Hungary. Figure A.13 shows a screenshot of a simple example.

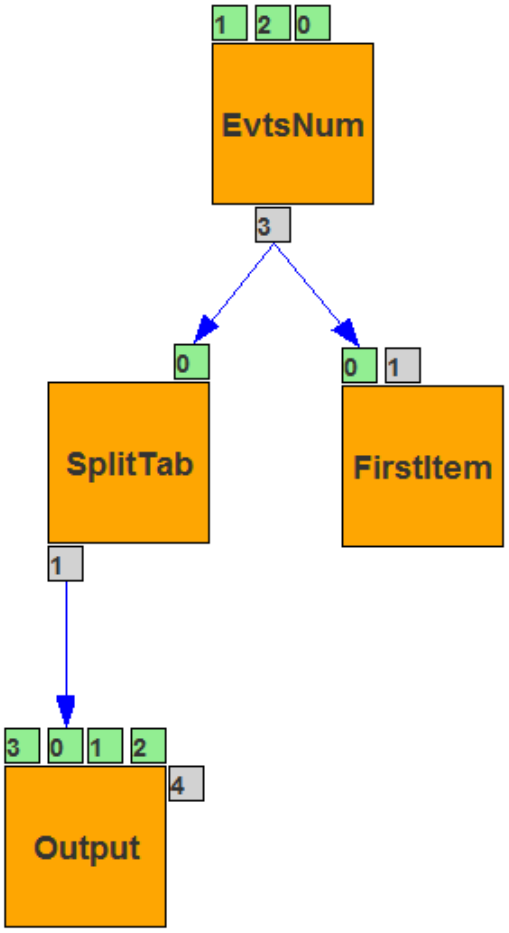


Figure A.13: Sample WS-PGRADE Workflow

[25] OPM: http://openprovenance.org/
[26] gUSE: http://www.guse.hu/

Interface WS-PGRADE is itself a web portal and a GUI for the underlying gUSE framework. Handy tools are provided for users to generate their own web portals dedicated to domain scientists and every scientific workflow can rather easily be bundled into a web service.

Model The scientific workflow model used in WS-PGRADE is quite transparently data-driven: workflows are composed graphically by creating "*jobs*" (*i.e.* activities), creating ports on them and then connecting output ports to input ports. A simple test reveals that the model is based on DAGs, since cycles prompt an error message.

Abstraction Level The level of abstraction of regular WS-PGRADE workflows is rather low, but the framework also handles a more abstract type of workflow called "*Template*" where constraints can be loosened at will and much indirection be inserted so that a given template might serve to create many different workflow instances.

APPENDIX B

CONCEPTUAL WORKFLOW META-MODEL

Figure B.4 describes the **Conceptual Workflow Model** in Unified Modeling Language (UML). In an attempt to improve legibility, we also provide three partial views:

- Figure B.1 focuses on the Conceptual part;
- Figure B.2 on the Abstract part; and
- Figure B.3 on the Semantic part.

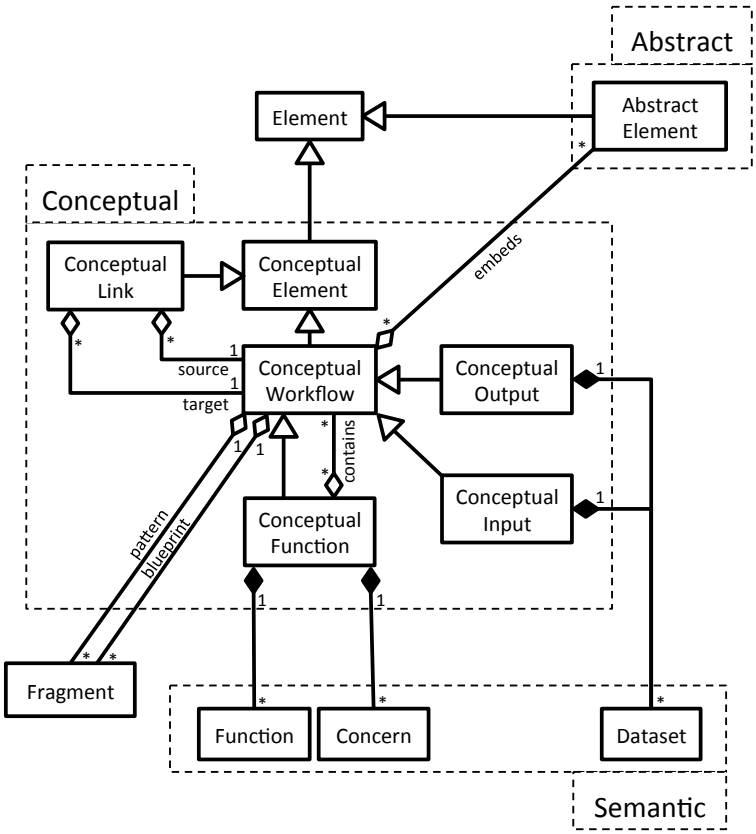


Figure B.1: Conceptual Workflow Meta-model - Conceptual Part

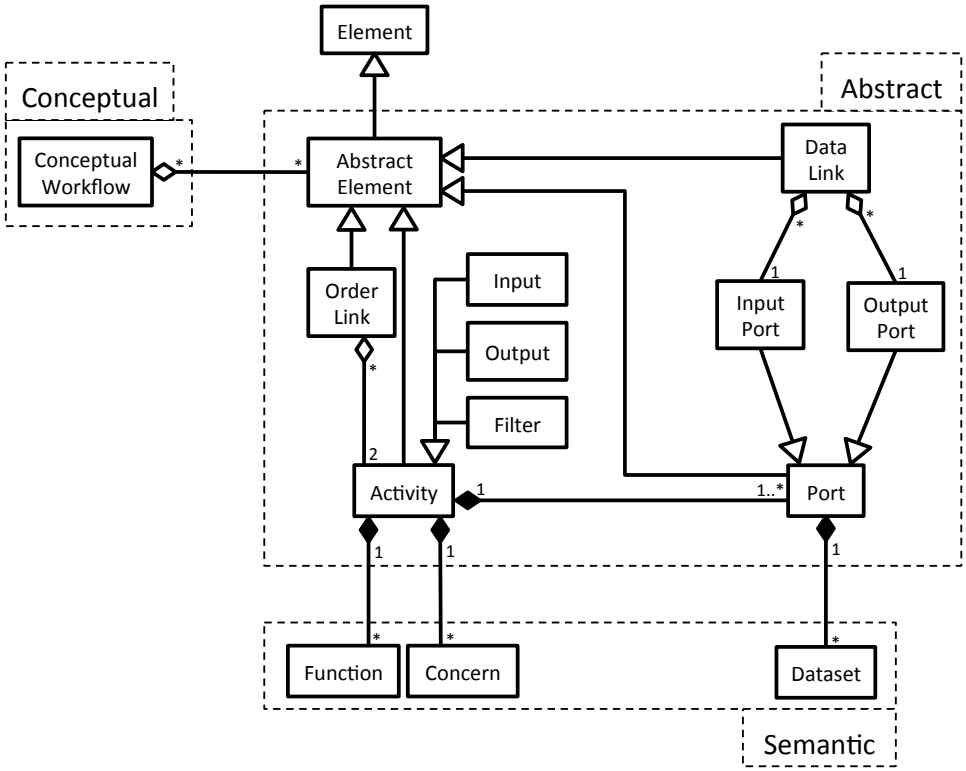


Figure B.2: Conceptual Workflow Meta-model - Abstract Part

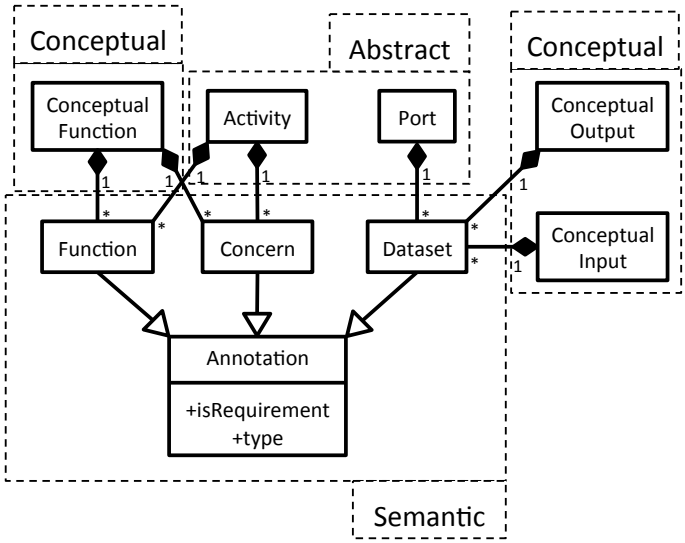


Figure B.3: Conceptual Workflow Meta-model - Semantic Part

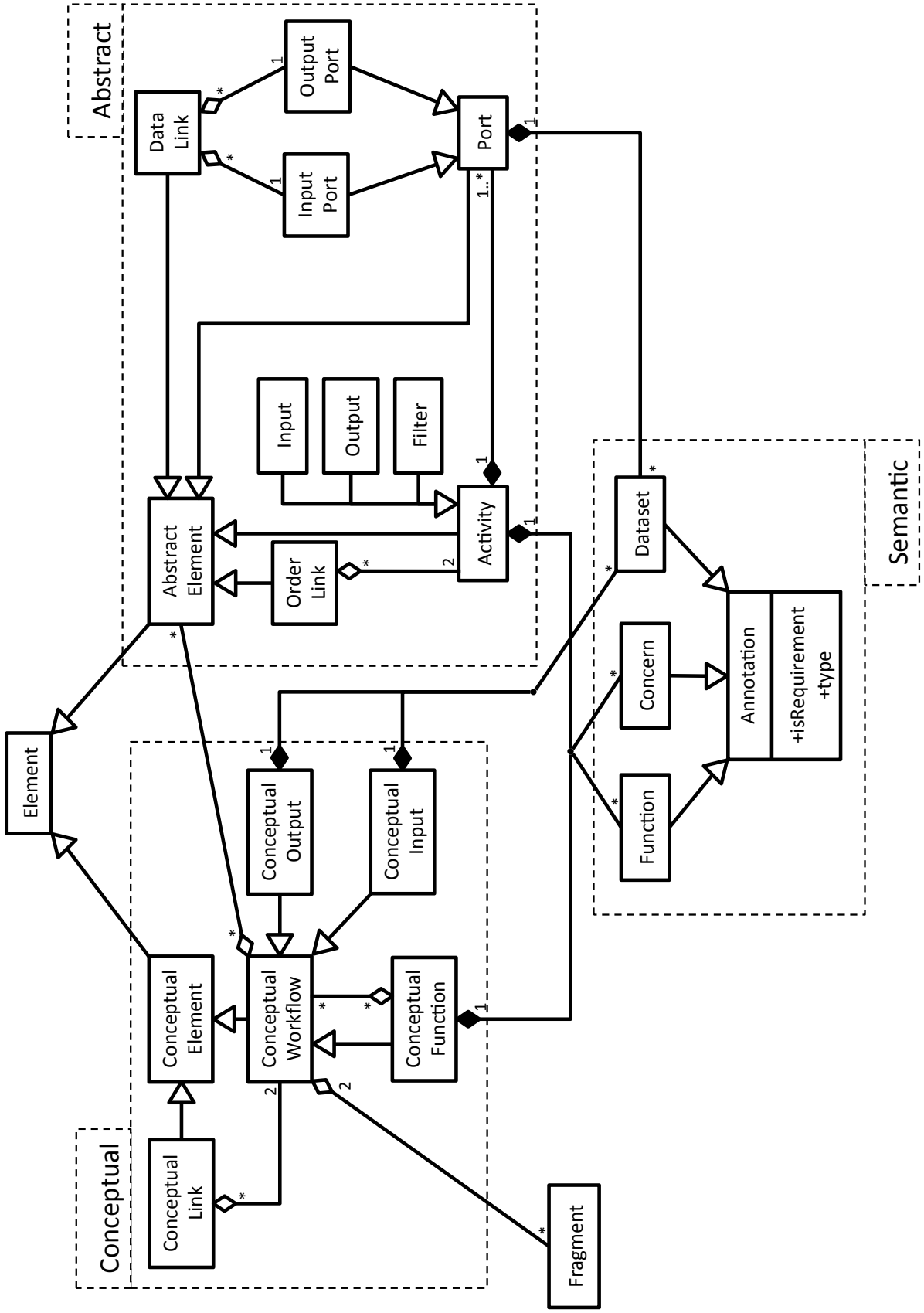


Figure B.4: Conceptual Workflow Meta-model

APPENDIX C

FRAGMENT TO SPARQL CONVERSION EXAMPLE

The following two listings illustrate the conversion from **Fragments** to SPARQL Protocol and RDF Query Language (SPARQL) `CONSTRUCT` queries, as explained in Section 4.2.2. The differences between the contents of those listings and what would be produced by the prototype (see Section 5.1) are (i) the prefixes in the SPARQL query, (ii) the manual ordering and spacing of triples, (iii) the omission of geometrical triples (setting height, width and position of elements) and `TO_REMOVE` comments (identifying deleted elements), all for the sake of legibility. Listing C.1 is the example **Fragment**, introduced in Section 3.4.1 and shown on Figure 3.10, in Turtle format.

Listing C.1: Fragment to SPARQL Conversion - Fragment Example

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix cowork: <http://www.i3s.unice.fr/~cerezo/cowork/latest/cowork.rdfs#> .
@prefix ex: <http://example.org/fragmentexample#> .
@prefix remote: <http://example.org/remoteontologyexample#> .

ex:Fragment1 rdf:type cowork:Fragment .
ex:Fragment1 cowork:hasBlueprint ex:Blueprint1 .
ex:Fragment1 cowork:hasPattern ex:Pattern1 .
ex:Fragment1 rdfs:label "Example" .

ex:Blueprint1 rdf:type cowork:ConceptualFunction .
ex:Blueprint1 cowork:contains ex:ConceptualFunction2 .
ex:Blueprint1 cowork:contains ex:ConceptualFunction3 .
ex:Blueprint1 rdfs:label "Example" .
ex:ConceptualFunction2 rdf:type cowork:ConceptualFunction .
ex:ConceptualFunction2 rdfs:label "Log␣Status" .
ex:ConceptualFunction2 cowork:hasRequirement ex:Function1 .
ex:Function1 rdf:type cowork:Function .
ex:Function1 rdf:type remote:Log .
ex:ConceptualFunction3 rdf:type cowork:ConceptualFunction .
ex:ConceptualFunction3 rdfs:label "Step" .
ex:ConceptualLink2 cowork:hasTarget ex:ConceptualFunction2 .
ex:ConceptualLink2 rdf:type cowork:ConceptualLink .
ex:ConceptualLink2 rdfs:label "linkbefore" .
ex:ConceptualLink3 cowork:hasSource ex:ConceptualFunction2 .
ex:ConceptualLink3 cowork:hasTarget ex:ConceptualFunction3 .
ex:ConceptualLink3 rdf:type cowork:ConceptualLink .

ex:Pattern1 rdf:type cowork:ConceptualFunction .
ex:Pattern1 cowork:contains ex:ConceptualFunction1 .
ex:Pattern1 rdfs:label "Example" .
ex:ConceptualFunction1 rdf:type cowork:ConceptualFunction .
ex:ConceptualFunction1 rdfs:label "Step" .
ex:ConceptualFunction1 cowork:hasRequirement ex:Concern1 .
ex:Concern1 rdf:type cowork:Concern .
ex:Concern1 rdf:type remote:CriticalStep .
ex:ConceptualLink1 cowork:hasTarget ex:ConceptualFunction1 .
ex:ConceptualLink1 rdf:type cowork:ConceptualLink .
ex:ConceptualLink1 rdfs:label "linkbefore" .
```

Listing C.2 is the result of converting the **Fragment** of the previous listing into a SPARQL `CONSTRUCT` query, which is the first step of the **Weaving** process.

Listing C.2: Fragment to SPARQL Conversion - Result Query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cowork: <http://www.i3s.unice.fr/~cerezo/cowork/latest/cowork.rdfs#>
PREFIX ex: <http://example.org/fragmentexample#>
PREFIX remote: <http://example.org/remoteontologyexample#>

CONSTRUCT {
  ?Example rdf:type cowork:ConceptualFunction .
  ?Example cowork:contains _:LogStatus .
  ?Example cowork:contains ?Step .
  _:LogStatus rdf:type cowork:ConceptualFunction .
  _:LogStatus rdfs:label "Log Status" .
  _:LogStatus cowork:hasRequirement _:Function1 .
  _:Function1 rdf:type cowork:Function .
  _:Function1 rdf:type remote:Log .
  ?Step rdf:type cowork:ConceptualFunction .
  ?linkbefore cowork:hasTarget _:LogStatus .
  ?linkbefore rdf:type cowork:ConceptualLink .
  ?linkbefore rdfs:label "linkbefore" .
  _:ConceptualLink3 cowork:hasSource _:LogStatus .
  _:ConceptualLink3 cowork:hasTarget ?Step .
  _:ConceptualLink3 rdf:type cowork:ConceptualLink .
} WHERE {
  ?Example rdf:type cowork:ConceptualFunction .
  ?Example cowork:contains ?Step .
  ?Step rdf:type cowork:ConceptualFunction .
  ?Step cowork:hasRequirement ?Concern1 .
  ?Concern1 rdf:type cowork:Concern .
  ?Concern1 rdf:type remote:CriticalStep .
  ?linkbefore cowork:hasTarget ?Step .
  ?linkbefore rdf:type cowork:ConceptualLink .
}
```

To facilitate comparison, those same listings are shown side-by-side in an abbreviated form (so as to fit the page width) in Section 4.2.2.

APPENDIX D

TWO CONVERTERS CHAIN QUERY

Listing D.1 is the SPARQL Protocol and RDF Query Language (SPARQL) query to look for a chain of two converters to transform from a source type `X` to a target type `Y`, cf. Section 4.5.4. The same query is represented graphically on Figure D.1.

Listing D.1: `X` → ? → `Y` Two Converters Chain Query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX cowork: <http://www.i3s.unice.fr/~cerezo/cowork/latest/cowork.rdfs#>

SELECT ?activity1, ?activity2 WHERE {
  ?activity1 rdf:type cowork:Activity .
  ?activity2 rdf:type cowork:Activity .

  ?activity1 cowork:hasInput ?inputport1 .
  ?inputport1 cowork:hasSpecification ?inputdataset1 .
  {
    ?inputdataset1 rdf:type X .
  }
  UNION
  {
    ?inputdataset1 rdf:type ?inputtype1 .
    X rdfs:subClassOf ?inputtype1 .
  }

  ?activity1 cowork:hasOutput ?outputport1 .
  ?outputport1 cowork:hasSpecification ?outputdataset1 .
  ?outputdataset1 rdf:type ?outputtype1 .

  ?activity2 cowork:hasInput ?inputport2 .
  ?inputport2 cowork:hasSpecification ?inputdataset2 .
  {
    ?inputdataset2 rdf:type ?outputtype1 .
  }
  UNION
  {
    ?inputdataset2 rdf:type ?inputtype2 .
    ?outputtype1 rdfs:subClassOf ?inputtype2 .
  }

  ?activity2 cowork:hasOutput ?outputport2 .
  ?outputport2 cowork:hasSpecification ?outputdataset2 .
  {
    ?outputdataset2 rdf:type Y .
  }
  UNION
  {
    ?outputdataset2 rdf:type ?outputtype2 .
    Y rdfs:subClassOf ?outputtype2 .
  }
}
```

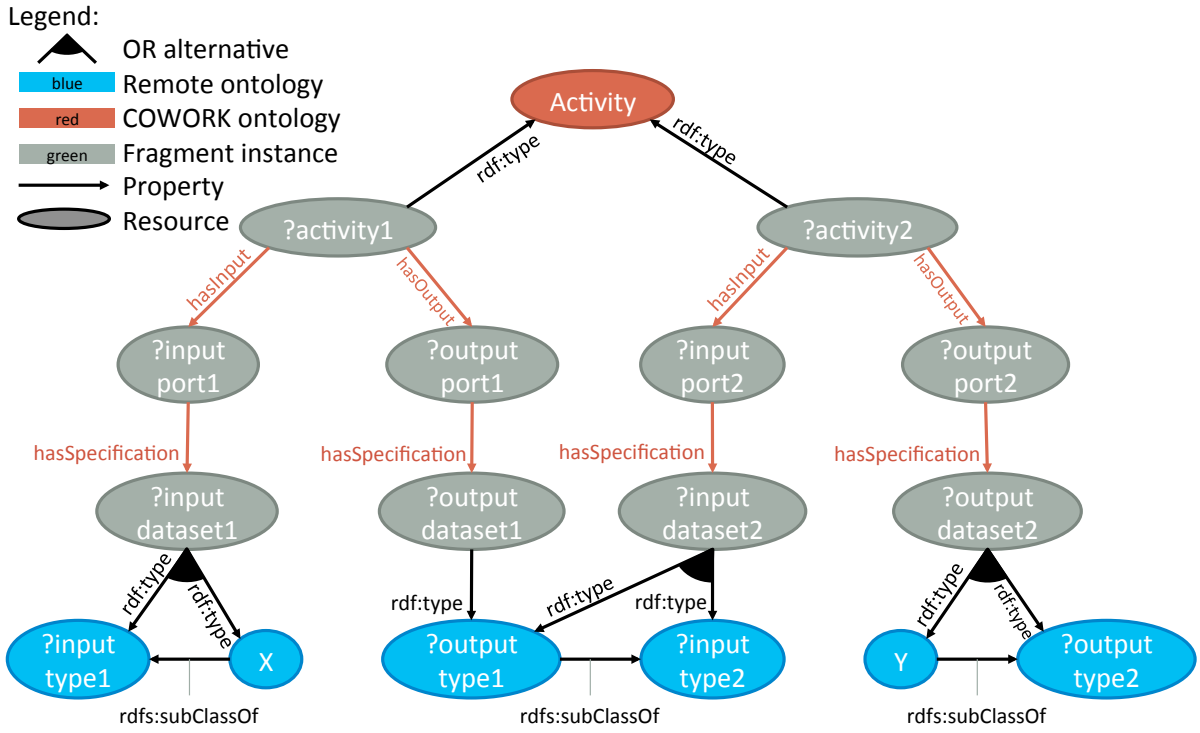


Figure D.1: X → ? → Y Two Converters Chain Query

APPENDIX E

CONVERSION TO T2FLOW

The following listings are the results of the **Conversion** process applied to the example **Intermediate Representations** given in Section 4.6, towards `t2flow`, the serialization format of Taverna [Missier 10a].

Specifically, Listing E.1 is the **Conversion** of the example shown on Figure 4.37a, which features only **Inputs** and **Outputs**; Listing E.2 of the one shown on Figure 4.38a, which features only **Activities**; and Listing E.3 of the one shown on Figure 4.39a, which features **Inputs**, **Outputs**, **Activities**, **Data Links** and one **Order Link**.

To improve legibility, Listing E.2 and Listing E.3 use the following abbreviations (reversing them is necessary before opening the files with `Taverna 2`):

- `cfg` → `net.sf.taverna.t2.workflowmodel.processor.activity.config`
- `bnsh` → `net.sf.taverna.t2.activities.beanshell`

Listing E.1: Conversion to t2flow - Inputs/Outputs Example

```
<workflow xmlns="http://taverna.sf.net/2008/xml/t2flow" producedBy="cowork" version="1">
  <dataflow role="top">
    <name>inputs/outputs example</name>
    <inputPorts>
      <port>
        <name>A</name><depth>0</depth><granularDepth>0</granularDepth><annotations/>
      </port>
      <port>
        <name>B</name><depth>0</depth><granularDepth>0</granularDepth><annotations/>
      </port>
      <port>
        <name>C</name><depth>0</depth><granularDepth>0</granularDepth><annotations/>
      </port>
    </inputPorts>
    <outputPorts>
      <port>
        <name>D_in0</name><depth>0</depth><granularDepth>0</granularDepth><annotations/>
      </port>
      <port>
        <name>D_in1</name><depth>0</depth><granularDepth>0</granularDepth><annotations/>
      </port>
    </outputPorts>
    <processors/>
    <conditions/>
    <datalinks/>
  </dataflow>
</workflow>
```

Listing E.2: Conversion to t2flow - Activities Example

```
<workflow xmlns="http://taverna.sf.net/2008/xml/t2flow" producedBy="cowork" version="1">
  <dataflow role="top">
    <name>activities example</name>
    <inputPorts/><outputPorts/>
    <processors>
      <processor>
        <name>P</name><inputPorts/><outputPorts/><annotations/>
        <activities>
          <activity>
            <raven>
              <group>net.sf.taverna.t2.activities</group>
              <artifact>beanshell-activity</artifact>
              <version>1.4</version>
            </raven>
            <class>bnsh.BeanshellActivity</class>
            <inputMap/><outputMap/>
            <configBean encoding="xstream">
              <bnsh.BeanshellActivityConfigurationBean>
                <inputs>
                  <cfg.ActivityInputPortDefinitionBean>
                    <name>in</name><depth>0</depth>
                    <translatedElementType>java.lang.String</translatedElementType>
                  </cfg.ActivityInputPortDefinitionBean>
                </inputs>
                <outputs>
                  <cfg.ActivityOutputPortDefinitionBean>
                    <name>out</name><depth>0</depth><granularDepth>0</granularDepth>
                  </cfg.ActivityOutputPortDefinitionBean>
                </outputs>
              </bnsh.BeanshellActivityConfigurationBean>
            </configBean>
          </activity>
        </activities>
        <dispatchStack/>
        <iterationStrategyStack><iteration><strategy/></iteration></iterationStrategyStack>
      </processor>
      <processor>
        <name>Q</name><inputPorts/><outputPorts/><annotations/>
        <activities>
          <activity>
            <raven>
              <group>net.sf.taverna.t2.activities</group>
              <artifact>beanshell-activity</artifact>
              <version>1.4</version>
            </raven>
            <class>bnsh.BeanshellActivity</class>
            <inputMap/><outputMap/>
            <configBean encoding="xstream">
              <bnsh.BeanshellActivityConfigurationBean>
                <inputs>
                  <cfg.ActivityInputPortDefinitionBean>
                    <name>in</name><depth>0</depth>
                    <translatedElementType>java.lang.String</translatedElementType>
                  </cfg.ActivityInputPortDefinitionBean>
                </inputs>
                <outputs>
                  <cfg.ActivityOutputPortDefinitionBean>
                    <name>out</name><depth>0</depth><granularDepth>0</granularDepth>
                  </cfg.ActivityOutputPortDefinitionBean>
                </outputs>
              </bnsh.BeanshellActivityConfigurationBean>
            </configBean>
          </activity>
        </activities>
        <dispatchStack/>
        <iterationStrategyStack><iteration><strategy/></iteration></iterationStrategyStack>
      </processor>
      <processor>
        <name>R</name><inputPorts/><outputPorts/><annotations/>
        <activities>
          <activity>
```

```
            <raven>
              <group>net.sf.taverna.t2.activities</group>
              <artifact>beanshell-activity</artifact>
              <version>1.4</version>
            </raven>
            <class>bnsh.BeanshellActivity</class>
            <inputMap/><outputMap/>
            <configBean encoding="xstream">
              <bnsh.BeanshellActivityConfigurationBean>
                <inputs>
                  <cfg.ActivityInputPortDefinitionBean>
                    <name>in</name><depth>0</depth>
                    <translatedElementType>java.lang.String</translatedElementType>
                  </cfg.ActivityInputPortDefinitionBean>
                </inputs>
                <outputs>
                  <cfg.ActivityOutputPortDefinitionBean>
                    <name>out</name><depth>0</depth><granularDepth>0</granularDepth>
                  </cfg.ActivityOutputPortDefinitionBean>
                </outputs>
              </bnsh.BeanshellActivityConfigurationBean>
            </configBean>
          </activity>
        </activities>
        <dispatchStack/>
        <iterationStrategyStack><iteration><strategy/></iteration></iterationStrategyStack>
      </processor>
      <processor>
        <name>S</name><inputPorts/><outputPorts/>
        <annotations/>
        <activities>
          <activity>
            <raven>
              <group>net.sf.taverna.t2.activities</group>
              <artifact>beanshell-activity</artifact>
              <version>1.4</version>
            </raven>
            <class>bnsh.BeanshellActivity</class>
            <inputMap/><outputMap/>
            <configBean encoding="xstream">
              <bnsh.BeanshellActivityConfigurationBean>
                <inputs>
                  <cfg.ActivityInputPortDefinitionBean>
                    <name>in0</name><depth>0</depth>
                    <translatedElementType>java.lang.String</translatedElementType>
                  </cfg.ActivityInputPortDefinitionBean>
                  <cfg.ActivityInputPortDefinitionBean>
                    <name>in1</name><depth>0</depth>
                    <translatedElementType>java.lang.String</translatedElementType>
                  </cfg.ActivityInputPortDefinitionBean>
                </inputs>
                <outputs>
                  <cfg.ActivityOutputPortDefinitionBean>
                    <name>out0</name><depth>0</depth><granularDepth>0</granularDepth>
                  </cfg.ActivityOutputPortDefinitionBean>
                  <cfg.ActivityOutputPortDefinitionBean>
                    <name>out1</name><depth>0</depth><granularDepth>0</granularDepth>
                  </cfg.ActivityOutputPortDefinitionBean>
                </outputs>
              </bnsh.BeanshellActivityConfigurationBean>
            </configBean>
          </activity>
        </activities>
        <dispatchStack/>
        <iterationStrategyStack><iteration><strategy/></iteration></iterationStrategyStack>
      </processor>
    </processors>
    <conditions/>
    <datalinks/>
  </dataflow>
</workflow>
```

Listing E.3: Conversion to t2flow - Links Example

```
<workflow xmlns="http://taverna.sf.net/2008/xml/t2flow" producedBy="cowork" version="1">
  <dataflow role="top">
    <name>links example</name>
    <inputPorts>
      <port>
        <name>A_out0</name><depth>0</depth><granularDepth>0</granularDepth><annotations/>
      </port>
      <port>
        <name>A_out1</name><depth>0</depth><granularDepth>0</granularDepth><annotations/>
      </port>
    </inputPorts>
    <outputPorts>
      <port><name>B_in0</name><annotations/></port>
      <port><name>B_in1</name><annotations/></port>
    </outputPorts>
    <processors>
      <processor>
        <name>P</name>
        <inputPorts>
          <port><name>in</name><depth>0</depth></port>
        </inputPorts>
        <outputPorts>
          <port><name>out</name><depth>0</depth><granularDepth>0</granularDepth></port>
        </outputPorts>
        <annotations/>
        <activities>
          <activity>
            <raven>
              <group>net.sf.taverna.t2.activities</group>
              <artifact>beanshell-activity</artifact>
              <version>1.4</version>
            </raven>
            <class>bnsh.BeanshellActivity</class>
            <inputMap><map from="in" to="in"/></inputMap>
            <outputMap><map from="out" to="out"/></outputMap>
            <configBean encoding="xstream">
              <bnsh.BeanshellActivityConfigurationBean>
                <inputs>
                  <cfg.ActivityInputPortDefinitionBean>
                    <name>in</name>
                    <depth>0</depth>
                    <translatedElementType>java.lang.String</translatedElementType>
                  </cfg.ActivityInputPortDefinitionBean>
                </inputs>
                <outputs>
                  <cfg.ActivityOutputPortDefinitionBean>
                    <name>out</name>
                    <depth>0</depth>
                    <granularDepth>0</granularDepth>
                  </cfg.ActivityOutputPortDefinitionBean>
                </outputs>
              </bnsh.BeanshellActivityConfigurationBean>
            </configBean>
          </activity>
        </activities>
        <dispatchStack/>
        <iterationStrategyStack><iteration><strategy/></iteration></iterationStrategyStack>
      </processor>
      <processor>
        <name>Q</name>
        <inputPorts>
          <port><name>in</name><depth>0</depth></port>
        </inputPorts>
        <outputPorts>
          <port><name>out</name><depth>0</depth><granularDepth>0</granularDepth></port>
        </outputPorts>
        <annotations/>
        <activities>
          <activity>
            <raven>
              <group>net.sf.taverna.t2.activities</group>
```

```
72              <artifact>beanshell-activity</artifact>
                <version>1.4</version>
74            </raven>
              <class>bnsh.BeanshellActivity</class>
76            <inputMap><map from="in" to="in"/></inputMap>
              <outputMap><map from="out" to="out"/></outputMap>
78            <configBean encoding="xstream">
                <bnsh.BeanshellActivityConfigurationBean>
80                <inputs>
                    <cfg.ActivityInputPortDefinitionBean>
82                    <name>in</name>
                      <depth>0</depth>
84                    <translatedElementType>java.lang.String</translatedElementType>
                    </cfg.ActivityInputPortDefinitionBean>
86                </inputs>
                  <outputs>
88                  <cfg.ActivityOutputPortDefinitionBean>
                      <name>out</name>
90                    <depth>0</depth>
                      <granularDepth>0</granularDepth>
92                  </cfg.ActivityOutputPortDefinitionBean>
                  </outputs>
94              </bnsh.BeanshellActivityConfigurationBean>
              </configBean>
96          </activity>
          </activities>
98        <dispatchStack/>
          <iterationStrategyStack><iteration><strategy/></iteration></iterationStrategyStack>
100     </processor>
        <processor>
102       <name>R</name>
          <inputPorts>
104         <port><name>in</name><depth>0</depth></port>
          </inputPorts>
106       <outputPorts>
            <port><name>out</name><depth>0</depth><granularDepth>0</granularDepth></port>
108       </outputPorts>
          <annotations/>
110       <activities>
            <activity>
112           <raven>
                <group>net.sf.taverna.t2.activities</group>
114             <artifact>beanshell-activity</artifact>
                <version>1.4</version>
116           </raven>
              <class>bnsh.BeanshellActivity</class>
118           <inputMap><map from="in" to="in"/></inputMap>
              <outputMap><map from="out" to="out"/></outputMap>
120           <configBean encoding="xstream">
                <bnsh.BeanshellActivityConfigurationBean>
122               <inputs>
                    <cfg.ActivityInputPortDefinitionBean>
124                   <name>in</name>
                      <depth>0</depth>
126                   <translatedElementType>java.lang.String</translatedElementType>
                    </cfg.ActivityInputPortDefinitionBean>
128               </inputs>
                  <outputs>
130                 <cfg.ActivityOutputPortDefinitionBean>
                      <name>out</name>
132                   <depth>0</depth>
                      <granularDepth>0</granularDepth>
134                 </cfg.ActivityOutputPortDefinitionBean>
                  </outputs>
136             </bnsh.BeanshellActivityConfigurationBean>
              </configBean>
138         </activity>
          </activities>
140       <dispatchStack/>
          <iterationStrategyStack><iteration><strategy/></iteration></iterationStrategyStack>
142     </processor>
        <processor>
144       <name>S</name>
```

```
        <inputPorts>
          <port><name>in</name><depth>0</depth></port>
        </inputPorts>
        <outputPorts>
          <port><name>out</name><depth>0</depth><granularDepth>0</granularDepth></port>
        </outputPorts>
        <annotations/>
        <activities>
          <activity>
            <raven>
              <group>net.sf.taverna.t2.activities</group>
              <artifact>beanshell-activity</artifact>
              <version>1.4</version>
            </raven>
            <class>bnsh.BeanshellActivity</class>
            <inputMap><map from="in" to="in"/></inputMap>
            <outputMap><map from="out" to="out"/></outputMap>
            <configBean encoding="xstream">
              <bnsh.BeanshellActivityConfigurationBean>
                <inputs>
                  <cfg.ActivityInputPortDefinitionBean>
                    <name>in</name>
                    <depth>0</depth>
                    <translatedElementType>java.lang.String</translatedElementType>
                  </cfg.ActivityInputPortDefinitionBean>
                </inputs>
                <outputs>
                  <cfg.ActivityOutputPortDefinitionBean>
                    <name>out</name>
                    <depth>0</depth>
                    <granularDepth>0</granularDepth>
                  </cfg.ActivityOutputPortDefinitionBean>
                </outputs>
              </bnsh.BeanshellActivityConfigurationBean>
            </configBean>
          </activity>
        </activities>
        <dispatchStack/>
        <iterationStrategyStack><iteration><strategy/></iteration></iterationStrategyStack>
      </processor>
    </processors>
    <conditions>
      <condition control="P" target="S"/>
    </conditions>
    <datalinks>
      <datalink>
        <sink type="processor"><processor>P</processor><port>in</port></sink>
        <source type="dataflow"><port>A_out0</port></source>
      </datalink>
      <datalink>
        <sink type="processor"><processor>Q</processor><port>in</port></sink>
        <source type="dataflow"><port>A_out1</port></source>
      </datalink>
      <datalink>
        <sink type="processor"><processor>R</processor><port>in</port></sink>
        <source type="processor"><processor>P</processor><port>out</port></source>
      </datalink>
      <datalink>
        <sink type="processor"><processor>S</processor><port>in</port></sink>
        <source type="processor"><processor>Q</processor><port>out</port></source>
      </datalink>
      <datalink>
        <sink type="dataflow"><port>B_in0</port></sink>
        <source type="processor"><processor>R</processor><port>out</port></source>
      </datalink>
      <datalink>
        <sink type="dataflow"><port>B_in1</port></sink>
        <source type="processor"><processor>S</processor><port>out</port></source>
      </datalink>
    </datalinks>
  </dataflow>
</workflow>
```

APPENDIX F

CONVERSION TO IWIR

The following listings are the results of the **Conversion** process applied to the example **Intermediate Representations** given in Section 4.6.4.2, towards `IWIR`, the pivot language of the interoperability platform SHIWA [Krefting 11].

All three listings show the result of **Conversion** of the example shown on Figure 4.41a:

- Listing F.1, illustrated on Figure 4.41b, corresponds to flat inputs, *i.e.* inputs with no extra depth levels compared to the specification of the target **Input Port**;
- Listing F.2, illustrated on Figure 4.41c, corresponds to inputs with 1 extra depth level each and a $A \odot B$ **Iteration Strategy**; and
- Listing F.2, illustrated on Figure 4.41d, corresponds to inputs with 1 extra depth level each and a $A \otimes B$ **Iteration Strategy**.

Listing F.1: Conversion to IWIR - Flat Inputs Example

```
<IWIR version="1.1" wfname="Flat␣Example" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="Flat_Example">
    <inputPorts>
      <inputPort name="A" type="string"/>
      <inputPort name="B" type="string"/>
    </inputPorts>
    <body>
      <task name="P" tasktype="P">
        <inputPorts>
          <inputPort name="in1" type="string"/>
          <inputPort name="in2" type="string"/>
        </inputPorts>
        <outputPorts>
          <outputPort name="out" type="string"/>
        </outputPorts>
      </task>
    </body>
    <outputPorts>
      <outputPort name="C" type="string"/>
    </outputPorts>
    <links>
      <link from="Flat_Example/A" to="P/in1"/>
      <link from="Flat_Example/B" to="P/in2"/>
      <link from="P/out" to="Flat_Example/C"/>
    </links>
  </blockScope>
</IWIR>
```

Listing F.2: Conversion to IWIR - Dot Example

```
<IWIR version="1.1" wfname="Dot␣Example" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="Dot_Example">
    <inputPorts>
      <inputPort name="A" type="collection/string"/>
      <inputPort name="B" type="collection/string"/>
    </inputPorts>
    <body>
      <parallelForEach name="P_dot_0">
        <inputPorts>
          <loopElements>
            <loopElement name="in1" type="collection/string"/>
            <loopElement name="in2" type="collection/string"/>
          </loopElements>
        </inputPorts>
        <body>
          <task name="P" tasktype="P">
            <inputPorts>
              <inputPort name="in1" type="string"/>
              <inputPort name="in2" type="string"/>
            </inputPorts>
            <outputPorts>
              <outputPort name="out" type="string"/>
            </outputPorts>
          </task>
        </body>
        <outputPorts>
          <outputPort name="out" type="collection/string"/>
        </outputPorts>
        <links>
          <link from="P_dot_0/in1" to="P/in1"/>
          <link from="P_dot_0/in2" to="P/in2"/>
          <link from="P/out" to="P_dot_0/out"/>
        </links>
      </parallelForEach>
    </body>
    <outputPorts>
      <outputPort name="C" type="collection/string"/>
    </outputPorts>
    <links>
      <link from="Dot_Example/A" to="P_dot_0/in1"/>
      <link from="Dot_Example/B" to="P_dot_0/in2"/>
      <link from="P_dot_0/out" to="Dot_Example/C"/>
    </links>
  </blockScope>
</IWIR>
```

Listing F.3: Conversion to IWIR - Cross Example

```
<IWIR version="1.1" wfname="Cross_Example" xmlns="http://shiwa-workflow.eu/IWIR">
  <blockScope name="Cross_Example">
    <inputPorts>
      <inputPort name="A" type="collection/string"/>
      <inputPort name="B" type="collection/string"/>
    </inputPorts>
    <body>
      <parallelForEach name="P_cross_0">
        <inputPorts>
          <inputPort name="in1" type="collection/string"/>
          <loopElements>
            <loopElement name="in2" type="collection/string"/>
          </loopElements>
        </inputPorts>
        <body>
          <parallelForEach name="P_cross_1">
            <inputPorts>
              <inputPort name="in2" type="string"/>
              <loopElements>
                <loopElement name="in1" type="collection/string"/>
              </loopElements>
            </inputPorts>
            <body>
              <task name="P" tasktype="P">
                <inputPorts>
                  <inputPort name="in1" type="string"/>
                  <inputPort name="in2" type="string"/>
                </inputPorts>
                <outputPorts>
                  <outputPort name="out" type="string"/>
                </outputPorts>
              </task>
            </body>
            <outputPorts>
              <outputPort name="out" type="collection/string"/>
            </outputPorts>
            <links>
              <link from="P_cross_1/in1" to="P/in1"/>
              <link from="P_cross_1/in2" to="P/in2"/>
              <link from="P/out" to="P_cross_1/out"/>
            </links>
          </parallelForEach>
        </body>
        <outputPorts>
          <outputPort name="out" type="collection/collection/string"/>
        </outputPorts>
        <links>
          <link from="P_cross_0/in1" to="P_cross_1/in1"/>
          <link from="P_cross_0/in2" to="P_cross_1/in2"/>
          <link from="P_cross_1/out" to="P_cross_0/out"/>
        </links>
      </parallelForEach>
    </body>
    <outputPorts>
      <outputPort name="C" type="collection/collection/string"/>
    </outputPorts>
    <links>
      <link from="Cross_Example/A" to="P_cross_0/in1"/>
      <link from="Cross_Example/B" to="P_cross_0/in2"/>
      <link from="P_cross_0/out" to="Cross_Example/C"/>
    </links>
  </blockScope>
</IWIR>
```

APPENDIX G

ONTOVIP URIS

The ontology of the Virtual Imaging Platform (VIP) project, OntoVIP, relies on other ontologies and imports multiple modules, which means it is not easy to guess the Uniform Resource Identifier (URI) of any OntoVIP resource based on its local name. For instance, `MR-simulation` comes from `VIP Simulation` and `MR-simulation-compatible-model` from `VIP Model`.

Note that the vast majority of resource **URIs are not URLs**: though they uniquely identify a resource, they cannot always be dereferenced and accessed with a web browser. Some of the URIs provided here will result in `404 File Not Found` errors or redirect to a generic web page, if used as URLs.

Here is the list, by alphebetical order, of ontology modules referred to in Chapter 5 and the corresponding URIs:

- DOLCE Particular
  → `http://neurolog.unice.fr/ontoneurolog/v3.0/dolce-particular.owl`
- OntoNeuroLOG DOLCE Extension
  → `http://neurolog.unice.fr/ontoneurolog/v3.0/ontoneurolog-extension-of-dolce.owl`
- OntoNeuroLOG Programs Software
  → `http://neurolog.unice.fr/ontoneurolog/v3.0/core-ontology-programs-software.owl`
- OntoNeuroLOG Dataset
  → `http://neurolog.unice.fr/ontoneurolog/v3.0/dataset.owl`
- OntoNeuroLOG MR Dataset Acquisition
  → `http://neurolog.unice.fr/ontoneurolog/v3.0/ontoneurolog-mr-dataset-acquisition.owl`
- VIP Model
  → `http://vip.creatis.insa-lyon.fr/ontovip/v1.0/vip-model.owl`
- VIP Simulated Data
  → `http://vip.creatis.insa-lyon.fr/ontovip/v1.0/vip-simulated-data.owl`
- VIP Simulation
  → `http://vip.creatis.insa-lyon.fr/ontovip/v1.0/vip-simulation.owl`

Here is the list, by alphabetical order, of resources referred to in Chapter 5 and in which of the aforementioned ontology modules they reside:

- CT-simulated-data → VIP Simulated Data
- CT-simulation → VIP Simulation
- CT-simulation-compatible-model → VIP Model
- directory → OntoNeuroLOG Programs Software
- echo-time → OntoNeuroLOG MR Dataset Acquisition
- file → OntoNeuroLOG Programs Software
- medical-image-simulation → VIP Simulation
- medical-image-simulation-object-model → VIP Model
- MR-simulated-image → VIP Simulated Data
- MR-simulation → VIP Simulation
- MR-simulation-compatible-model → VIP Model
- number → OntoNeuroLOG DOLCE Extension
- PET-list-mode-data → VIP Simulated Data
- PET-simulated-data → VIP Simulated Data
- PET-simulated-image → VIP Simulated Data
- PET-simulation → VIP Simulation
- PET-simulation-compatible-model → VIP Model
- PET-sinogram → VIP Simulated Data
- proton-density-weighted-MR-dataset → OntoNeuroLOG Dataset
- quality → DOLCE Particular
- repetition-time → OntoNeuroLOG MR Dataset Acquisition
- simulated-data → VIP Simulated Data
- T1-weighted-MR-dataset → OntoNeuroLOG Dataset
- T2-weighted-MR-dataset → OntoNeuroLOG Dataset
- US-simulated-image → VIP Simulated Data
- US-simulation → VIP Simulation
- US-simulation-compatible-model → VIP Model
- version-number → OntoNeuroLOG DOLCE Extension

To obtain the full URI of a resource, one simply has to concatenate the URI of the ontology module where the resource resides with the local name of the resource, with a hash character in between. For instance, quality resides in DOLCE Particular, therefore its full URI is:

http://neurolog.unice.fr/ontoneurology/v3.0/↩
dolce-particular.owl#quality

APPENDIX H

LICENSE

This work is licensed under **Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International**.

You are free to:

- **Share** copy and redistribute the material in any medium or format
- **Adapt** remix, transform, and build upon the material
- The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** You may not use the material for commercial purposes.
- **ShareAlike** If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- No additional restrictions You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices:

- You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.
- No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

For the exact legal terms of the license, please refer to:
`http://creativecommons.org/licenses/by-nc-sa/4.0/legalcode`

GLOSSARY

Abstract Element: Element of the abstract part of the **Conceptual Workflow Model**: either an **Activity**, a **Port**, a **Data Link** or a **Order Link**.
→ Section 3.2
→ Pages 8, 43, 47, 48, 51–53, 57, 73, 75, 110, 121, 145, 183, 191

Abstract Level: Intermediary level of abstraction between the **Conceptual Level** and the **Concrete Level**. Most scientific workflow frameworks handle scientific workflows at that level. It aligns with the level of Platform-Independent Models (PIMs) in the Model Driven Architecture (MDA).
→ Chapter 2
→ Pages 4–6, 15, 43, 47, 48, 57, 122, 144, 145, 149, 150, 153–155, 157–159

Abstract Workflow: Workflow described at an intermediate abstraction level between the simulation it represents and actual enactment. Most scientific workflow frameworks handle scientific workflows at that level.
→ Pages 43, 52, 57, 84, 94, 106, 114, 115, 117, 118, 120–122, 125, 128, 129, 144, 161, 186, 190, 197

Activity: Element of the **Conceptual Workflow Model** which models executable artifacts such as web services, grid jobs and legacy programs.
→ Section 3.2.1
→ Pages 49–51, 53, 58, 84, 86–91, 95–97, 99–102, 104, 107, 111, 123–126, 128, 130, 131, 138, 139, 142, 145, 146, 170, 183, 186, 188–190, 193, 194

Acyclic Graph: A directed graph without cycles: no matter the starting vertex, it is impossible to get back to it by following the edges. Since the number of vertices is finite, it follows that the number of paths between two vertices in an acyclic graph is always finite.
→ Pages 186

AGWL: Scientific workflow language underlying ASKALON. It is compatible with eXtensible Markup Language (XML) and control-driven.
→ [Fahringer 05]
→ `http://www.dps.uibk.ac.at/projects/agwl/`
→ Pages 147, 152, 184

Annotation: Element of the **Conceptual Workflow Model** which models either a **Conceptual Element** or **Abstract Element** and links it with a concept defined in an external domain, technical or non-functional ontology. Defined by its **Type**, **Role** and **Meaning**.
→ Section 3.3
→ Pages 51–55, 58, 61, 62, 68, 73–78, 80–82, 86, 87, 91, 92, 109–111, 123, 132, 138, 139, 142, 145, 186–188, 191, 192, 194–197

AOP (Aspect-Oriented Programming): Software development paradigm focusing on cross-cutting concerns: the base process is defined separately from aspects which are woven into it automatically.
→ Pages 20, 21, 23, 24, 59

API (Application Programming Interface): A protocol meant to allow automated communication between programs.
→ Pages 11, 16, 40, 114, 153, 155, 198

ASKALON: Free (for educational and research purposes) scientific workflow framework developed mainly at the University of Innsbruck, Austria. It uses Unified Modeling Language (UML) and AGWL to model scientific workflows, and targets Cloud and Grid environments.
→ [Fahringer 07]
→ `http://www.dps.uibk.ac.at/projects/askalon/`
→ Pages 14, 147–149, 155, 183

Bipartite Graph: A graph whose vertices can be classified in two groups with no internal edges in either.
→ Pages 193

Blueprint: **Conceptual Workflow** which is part of a **Fragment** and represents how the **Fragment** will be woven, *i.e.* what the **Weaving** mechanism will build in the base process to weave the **Fragment**.
→ Pages 54, 55, 59, 60, 63–66, 68, 69, 72, 77, 78, 80, 82, 83, 130, 145, 188

BPEL (Business Process Execution Language): Executable language defined by the Organization for the Advancement of Structured Information Standards (OASIS) to specify business processes that are based on web services. With time, it has emerged as the *de facto* standard in the field of business workflows.
→ `https://www.oasis-open.org/committees/wsbpel/`
→ Pages 10, 23

Business Process: Structured set of tasks meant to achieve a business goal such as providing a service or making a product.
→ Pages 10, 17, 23, 24, 184

Business Workflow Framework: A workflow framework dedicated to one or more business workflow model(s).
→ Pages 10

Business Workflow: A workflow meant to formalize and/or enact a business process.
→ Pages 10, 11, 17, 23, 24, 184

CIM (Computation-Independent Model): Class of models defined by the MDA and based on the abstraction level: computation-independent models are so high-level as to not be tied to a specific implementation method or infrastructure. They are designed to be easy for domain experts to understand, design and manipulate. However, they must be transformed into lower-level models to be used in practice.
→ Pages 5, 25, 26, 28, 29, 185, 191

Composition: Composition is the name of the part of the **Mapping** process dedicated to helping users compose their **Conceptual Workflow** in order to transform it into an **Intermediate Representation**. It consists in suggesting links between unattached **Input Ports** and compatible predecessing **Output Ports** and looking for converters when mismatches are detected.
→ Section 4.5
→ Pages 58, 84, 110, 111, 130, 138, 142, 145

Conceptual Element: Element of the conceptual part of the **Conceptual Workflow Model**: either a **Conceptual Workflow**, **Conceptual Function**, **Conceptual Input**, **Conceptual Output** or **Conceptual Link**.
→ Section 3.1
→ Pages 43, 44, 46, 51–53, 57, 110, 183

Conceptual Function: Element of the **Conceptual Workflow Model** which models scientific process steps at the conceptual level of abstraction.
→ Section 3.1.1
→ Pages 43, 44, 46, 53, 55, 62, 63, 66–70, 72–74, 81, 82, 84, 88, 123, 124, 126, 128, 131, 133–135, 146, 185, 188, 193, 196

Conceptual Input: Element of the **Conceptual Workflow Model** which models the input data of a simulation at the conceptual level of abstraction.
→ Section 3.1.1
→ Pages 43, 44, 46, 48, 53, 58, 60, 72, 74, 75, 81–83, 88, 185

Conceptual Level: Level of abstraction at which simulations are conceived by scientists, in their own domain(s). It aligns with the level of Computation-Independent Models (CIMs) in the MDA.
→ Chapter 2
→ Pages 4–6, 8, 29, 43, 57, 122, 144, 150, 155, 161, 183, 199

Conceptual Link: Element of the **Conceptual Workflow Model** which models dependencies between components of a simulation at the conceptual level of abstraction.
→ Section 3.1.4
→ Pages 43, 44, 47, 48, 55, 60, 62, 66, 73, 75, 87, 88, 135, 185

Conceptual Output: Element of the **Conceptual Workflow Model** which models the output products of a simulation at the conceptual level of abstraction.
→ Section 3.1.1
→ Pages 43, 44, 46, 48, 53, 72, 74, 75, 81–83, 88, 185

Conceptual Workflow Model: The Workflow model we propose to formalize simulations at a high level of abstraction that is computation-independent, as well as enable the computer-assisted transformation to executable workflows.
→ Chapter 3
→ Pages 29, 41, 43, 48, 49, 51, 57, 62, 66, 76, 94, 102, 107, 109, 114, 129, 144, 145, 163, 183, 185, 186, 189, 193–195

Conceptual Workflow: Instance of the **Conceptual Workflow Model**.
→ Chapter 3
→ Pages 8, 30, 41, 43, 44, 46–49, 52, 54, 55, 57–59, 62–68, 72–77, 80–82, 84, 87, 88,

94, 107, 109, 114, 122, 123, 129, 130, 132, 137, 138, 142, 144–146, 184, 185, 187, 188, 190–198

Concern: One of the three types of **Meaning** for an **Annotation**, which means it represents a non-functional criterion.
→ Section 3.3.3
→ Pages 52, 53, 76–78, 82, 84, 92, 94, 111, 122, 123, 128, 131, 139

Concrete Level: Level of abstraction at which scientific workflows are enacted. It aligns with the level of Platform-Specific Models (PSMs) in the MDA.
→ Chapter 2
→ Pages 4, 5, 150, 155, 156, 183

Control-driven Model: Graph-based scientific workflow model where edges represent control flow.
→ Section 2.1.2.3
→ Pages 12, 14, 102, 104, 107, 151, 152, 156

Conversion: Conversion is the name of the second step of the **Transformation Process**. It is an automated transformation from the **Intermediate Representation** to a target external abstract workflow language.
→ Section 4.6
→ Pages 6, 8, 29, 57, 94, 95, 100–104, 106, 107, 110, 111, 130, 139, 144, 146, 170, 176, 197

COWORK (COnceptual WORKflow): the **Conceptual Workflow Model** captured as an RDFS-based ontology.
Available at `http://www.i3s.unice.fr/~cerezo/cowork/latest/cowork.rdfs`
→ Pages 51, 52, 63

Cross Product: Binary operator combining **Input Ports** and used in **Iteration Strategies** to specify that each piece of data received on an **Input Port** must be paired up with every possible combination of the pieces of data received on the other **Input Ports**. $(a_1, a_2) \otimes (b_1, b_2) = \{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$
→ Pages 50

Cross-cutting concern: Non-functional concern (*i.e.* which is not integral to the basic functionalities of a process, *e.g.* Quality of Service (QoS) concerns) which impacts the structure of the process or is replicated in multiple locations in the process.
→ Pages 184

Cyclic Graph: Synonym of directed graph, used to emphasize the difference with acyclic graphs that do not allow cycles.
→ Pages 187, 192

DAG (Directed Acyclic Graph): See directed graph, acyclic graph and graph.
→ Pages 12, 17, 153, 155, 157, 160, 162

Data Link: Element of the **Conceptual Workflow Model** which models data flow between two **Activities**: data is transferred from the source to the target and data reception fires the target.

→ Section 3.2.3
→ Pages 49, 51, 86–88, 91, 92, 95, 97, 99, 101, 102, 123, 129, 138, 139, 170, 183

Data-driven Model: Graph-based scientific workflow model where edges represent data flow.
→ Section 2.1.2.3
→ Pages 12, 13, 15, 149, 150, 152, 156

Dataset: One of the three types of **Meaning** for an **Annotation**, which means it qualifies data content or data format.
→ Section 3.3.3
→ Pages 52, 53, 82, 84, 94, 132

DAX: Scientific workflow language underlying Pegasus. It is compliant with XML and serves as the primary interface with the enactor.
→ `http://pegasus.isi.edu/wms/docs/latest/creating_workflows.php`
→ Pages 155

DCG (Directed Cyclic Graph): See directed graph, cyclic graph and graph.
→ Pages 12, 17, 43, 147, 149, 152, 155, 158, 189

DCI (Distributed Computing Infrastructure): Set of distributed (and often heterogeneous) computing resources used in concert to execute programs, *e.g.* service platforms, supercomputers, computing grids, clouds.
→ Pages 4, 5, 12, 14, 43, 50, 107, 144, 147, 149, 150, 155, 157, 161, 187, 188, 193, 199

Directed Graph: A graph whose links are called edges and have a direction: they go from a source to a target and are most often represented by arrows.
→ Pages 2, 3, 12, 43, 44, 183, 186–188, 192

Discovery: Discovery is the name of the part of the **Mapping** process dedicated to helping users find **Fragments** relevant to the **Conceptual Workflow** at hand. It consists in matching the **Annotations** of that **Conceptual Workflow** against that of **Fragments** in the knowledge base, ranking the **Fragments** that matched by order of relevance and presenting them to the user for them to pick the one that will be woven into the workflow.
→ Section 4.4
→ Pages 58, 76, 89, 110, 111, 130, 133–135, 142, 145, 191, 194

Distributed Algorithm: Algorithm designed to be run in parallel on multiple inter-connected processing units.
See also Distributed Computing Infrastructure (DCI).
→ Pages 2–4

DOLCE: Foundational ontology meant to serve as a basis to design and/or align specialized ontologies.
→ [Gangemi 02]
→ Pages 32, 113

Dot Product: Binary operator combining **Input Ports** and used in **Iteration Strategies** to specify that the pieces of data received on the **Input Ports** must be paired up according to their order of arrival. $(a_1, a_2) \odot (b_1, b_2) = \{(a_1, b_1), (a_2, b_2)\}$
→ Pages 50

Enactor: Program deploying scientific workflows on DCIs and managing their execution.
→ Pages 5, 11–13, 15, 50, 84, 102, 104, 147, 151, 154, 155, 187, 190, 197

Erasing: Erasing is a mechanism used in the **Mapping** step of the **Transformation Process** to remove a **Conceptual Function** and redistribute its sub-workflows and annotations automatically. It is especially useful after node-bound **Weaving**, as explained in 4.3.2.
→ Section 4.3.2
→ Pages 58, 59, 63, 70, 72, 73, 76, 110

Explicit Port: **Port** that is explicitly declared in the description of the executable artifact underlying the **Activity** the **Port** is associated to.
→ Section 3.2.1
→ Pages 49, 51

FIELD-II: UltraSonography (US) simulator seminally included in the Virtual Imaging Platform (VIP) platform.
→ [Jensen 04]
→ `http://field-ii.dk/`
→ Pages 114

Filter: Specialized **Activity** which models conditionals.
→ Section 3.2.2
→ Pages 49, 51

FOP (Feature-Oriented Programming): Software development paradigm defining programs as a composition of features, *i.e.* sets of formally defined sets of functionalities.
→ Pages 21, 196

Fragment: Combination of a **Pattern** and **Blueprint** which can be merged into a **Conceptual Workflow** using the **Weaving** mechanism.
→ Section 3.4
→ Pages 24, 54, 55, 58–70, 72, 76–78, 80–84, 89, 110, 111, 130–135, 137, 142, 145, 166, 167, 184, 187, 191, 193, 194, 198

Function: One of the three types of **Meaning** for an **Annotation**, which means it represents a domain process step.
→ Section 3.3.3
→ Pages 49, 52, 53, 55, 76–78, 82, 84, 92, 94, 111, 112, 122, 123, 126, 130–133, 139, 189

Galaxy: Open-source web-based scientific workflow framework developed mainly at the universities of Penn State and Emory, USA.
→ [Goecks 10]
→ `http://galaxyproject.org/`
→ Pages 149

Graph: A set of objects represented by vertices (also known as nodes or points), pairs of which are connected by links (also known as arcs and generally called edges when directed) representing some relation between the objects.
See also directed graph.
→ Pages 3, 12, 44, 58, 63, 65–68, 71, 72, 78, 80, 103, 106, 184, 186, 187, 192

Guard: Logical condition associated to a **Function**. It is evaluated to determine along which branch data should be transferred.
→ Section 3.2.2
→ Pages 49

GUI (Graphical User Interface): User interface using image-based input-output methods (*e.g.* mouse, touch-screen) rather than text-based methods (*i.e.* command-line interface).
→ Pages 3, 11, 16, 109, 111, 147, 150–156, 158, 161, 162, 199

GWENDIA: Scientific workflow language underlying MOTEUR. It is compatible with XML and has a data-driven Directed Cyclic Graph (DCG) core extended with select control constructs and mechanisms inspired by array programming.
→ [Montagnat 09]
→ `http://gwendia.i3s.unice.fr/`
→ Pages 94, 95, 100, 114, 122, 130, 152, 154, 155

GWES: Scientific workflow framework whose language, called GWorkflowDL, is based on petri nets. It is developed at the Fraunhofer Institute for Open Communication Systems, Germany, and is free of charge for a Licensee's own scientific or educational purposes.
→ [Neubauer 05]
→ `http://gridworkflow.org/kwfgrid/gwes-web/`
→ Pages 12, 150, 189

GWorkflowDL: Scientific workflow language underlying GWES. It is compliant with XML, based on petri nets and is very flexible when it comes to how much information is specified about operations, from nothing to full instantiation, going through references to a registry and lists of candidates.
→ [Alt 05]
→ `http://gridworkflow.org/kwfgrid/gworkflowdl/docs/index.html`
→ Pages 150, 189

Hybrid Model: Graph-based scientific workflow model where edges represent either data flow or control flow.
→ Section 2.1.2.3
→ Pages 12, 13, 15, 102, 153

Immediate Parent Workflow: workflow whose relation with the considered sub-workflow is defined directly, rather than by transitivity.
→ Section 3.1.1
→ Pages 46, 48

Immediate Sub-workflow: workflow whose relation with the considered parent workflow is defined directly, rather than by transitivity.
→ Section 3.1.1
→ Pages 46

Implicit Port: **Port** that can be modeled in the **Conceptual Workflow Model**, but is not explicitly declared in the description of the executable artifact underlying the **Activity** the **Port** is associated to.
→ Section 3.2.1
→ Pages 49, 51

In-silico Experiment: Synonym of simulation that stems from and is especially used in the domain of life sciences.
→ Pages 1

Input Port: **Port** representing an argument of the underlying executable artifact represented by the **Activity** the **Port** is associated to.
→ Section 3.2.1
→ Pages 49, 50, 58, 84, 86–89, 91, 96, 97, 100, 101, 104, 123, 129, 138, 176, 185–187, 190

Input: Specialized **Activity** which models data sources.
→ Section 3.2.2
→ Pages 49, 51, 95–97, 99, 100, 102, 104, 124, 137, 170

Intermediate Representation: Fully mapped **Conceptual Workflow**, ready to be converted to an abstract workflow. It is called intermediate, because it stands right between the two steps of the **Transformation Process**.
→ Pages 6, 8, 29, 57, 59, 84, 94, 96, 97, 100–103, 106, 122, 124–126, 129, 130, 139, 145, 170, 176, 185, 186, 191

IPL (Imperative Programming Language): programming language defining precise commands for the computer to perform, such as Fortran and C.
→ Pages 14, 147

Iteration Strategy: Composition of **Input Ports** and operators that defines how input data must be combined to be fed to an **Activity**.
→ Section 3.2.4
→ Pages 50, 97, 101, 104, 107, 176, 186, 187

IWIR: Pivot scientific workflow language used on the SHIWA platform to enable fine-grained interoperability.
→ [Plankensteiner 11]
→ `http://www.shiwa-workflow.eu/wiki/-/wiki/Main/IWIR`
→ Pages 94, 102–104, 107, 146, 155, 156

Java CoG Kit: Free scientific workflow framework integrated into the Globus Toolkit[1]. It is developed in many institutions including but far from limited to the Argonne National Library, USA and the Indiana University, USA. The underlying language is Karajan.
→ [von Laszewski 01]
→ `http://wiki.cogkit.org/wiki/Main_Page`
→ Pages 151, 190

Karajan: Control-driven scientific workflow language used as inner language by the Java CoG Kit and as enactor by Swift.
→ [von Laszewski 07]
→ `http://wiki.cogkit.org/wiki/Karajan`
→ Pages 151, 156, 190, 197

[1] Globus Toolkit: → `http://www.globus.org/toolkit/`

Kepler: Open-source scientific workflow framework based on actor-oriented design framework Ptolemy II[2]. The project has contributors from plenty of institutions and originated from the University of Davis, USA, the University of Santa Barbara, USA and the University of San Diego, USA. The underlying language is MoML (the same as Ptolemy II).
→ [Ludäscher 06]
→ `https://kepler-project.org/`
→ Pages 12, 22, 152, 153, 192

KNIME: Scientific workflow framework mainly developed at the University of Konstanz, Germany, and open-source platform around which the eponymous company provides support and services. Its name stands for KoNstanz Information MinEr.
→ [Berthold 07]
→ `http://www.knime.org/`
→ Pages 153, 154

Knowledge Base: Semantic database or in-memory semantic graph, generally composed of interconnected triples.
See also triple store.
→ Pages 33, 54, 58, 63, 72, 76, 77, 80, 83, 84, 86, 88, 89, 91, 107, 110, 111, 130, 132, 137, 142, 144–146, 187, 191

Knowledge Engineering: Engineering discipline and research domain aiming at automated use and/or production of knowledge.
→ Pages 30

Life Sciences: Research domain dedicated to the study of living organisms, including, for instance, the sciences of biology, physiology and biochemistry.
→ Pages 1, 190

Mapping: Mapping is the name of the first step of the **Transformation Process**. It consists in finding suitable **Abstract Elements** and/or sub-workflows to implement the simulation modeled by the **Conceptual Workflow**. The result is called **Intermediate Representation**.
→ Section 4.1
→ Pages 6, 8, 29, 41, 52, 57–59, 71, 76, 77, 84, 86, 94, 106, 109, 130, 142, 144–146, 185, 187, 188, 192, 197, 198

Matching: Matching is the first part of the **Discovery** process. It consists in querying the knowledge base to the **Annotations** of that **Conceptual Workflow** against that of **Fragments** in the knowledge base, ranking the **Fragments** that matched by order of relevance and presenting them to the user for them to pick the one that will be woven into the workflow.
→ Section 4.4.2
→ Pages 77, 78, 80, 194

MDA (Model Driven Architecture): Model Driven Engineering (MDE) approach launched by the Object Management Group (OMG) which, among many things, classifies software models into three abstraction levels: PSM, PIM and CIM. → `http://www.omg.org/`

[2]Ptolemy II: → `http://ptolemy.berkeley.edu/ptolemyII/`

mda/
→ Pages 5, 24, 25, 29, 183–186, 192, 194

MDE (Model Driven Engineering): Approach to software development that focuses on the creation and management of high-level domain models on which development is based, in order to leverage domain experts' knowledge and ease communication between system designers.
→ Pages 11, 22, 24, 26, 28, 29, 191

Meaning: Defining feature of an **Annotation** that categorizes the concept it links to, from the viewpoint of the **Conceptual Workflow**.
→ Section 3.3.3
→ Pages 51, 52, 54, 74, 84, 183, 186–188

Merging: Merging is a mechanism used in the **Mapping** step of the **Transformation Process** to combine two **Conceptual Workflows** into one. It is especially useful after link-bound **Weaving**, as explained in 4.2.6.
→ Section 4.3.1
→ Pages 58, 59, 63, 70, 72, 73, 76, 110

MoC (Model of Computation): Describes how the components of a system interact to perform the high-level goals. Like the actor-oriented design framework it is based on, Kepler defines scientific workflow graphs and models of computation separately, which gives it flexible execution semantics.
→ Pages 152

MoML: The Modeling Markup Language is an XML dialect Kepler inherits from the actor-oriented design framework it is based on.
→ [Lee 00]
→ http://ptolemy.eecs.berkeley.edu/projects/summaries/00/moml.html
→ Pages 152, 191

MOTEUR: Open-source scientific workflow manager developed mainly at the I3S Laboratory, France. It is dedicated to efficient enactment on computing grids and uses the GWENDIA [Montagnat 09] language.
→ [Glatard 08]
→ http://modalis.i3s.unice.fr/softwares/moteur/
→ Pages 15, 50, 94, 139, 154, 155, 189

Nested Directed Cyclic Graph: See nested graph, directed graph, cyclic graph and graph.
→ Pages 17, 43

Nested Graph: A graph whose vertices can themselves be subgraphs.
→ Pages 44, 192

OMG (Object Management Group): International computer industry consortium created to define standards in the field of object-oriented software development. UML and the MDA are among their most well-known standards.
→ www.omg.org
→ Pages 25, 26, 191

OPM (Open Provenance Model): as the name indicates, it is a system-agnostic standard to describe and exchange scientific workflow provenance data. It resulted from the Provenance Challenge series and many scientific workflow frameworks can export it now.
→ http://openprovenance.org/
→ http://twiki.ipaw.info/bin/view/Challenge/
→ Pages 3

Order Link: Element of the **Conceptual Workflow Model** which models control flow between two **Activities**: when the source terminates, control is passed over to the target. There might be a delay between the end of the source and the beginning of the target, depending on the enactment strategy and available ressources, but the target never fires before the source ends.
→ Section 3.2.3
→ Pages 49, 51, 87, 95, 97, 99, 101, 102, 170, 183

Output Port: **Port** representing a product of the underlying executable artifact represented by the **Activity** the **Port** is associated to.
→ Section 3.2.1
→ Pages 49, 84, 86–90, 96, 97, 100, 123–126, 129, 185

Output: Specialized **Activity** which models data sinks.
→ Section 3.2.2
→ Pages 49, 51, 95–97, 99–101, 103, 104, 137, 170

OWL (Web Ontology Language): Language specified and maintained by the World Wide Web Consortium (W3C) to formally define ontologies.
→ http://www.w3.org/TR/owl-overview/
→ Pages 32, 38, 41, 194

Parent workflow: workflow or **Conceptual Function** which contains one or more workflows or **Conceptual Workflows**, relatively called its sub-workflows.
→ Section 3.1.1
→ Pages 24, 46–48, 189, 196

Pattern: **Conceptual Workflow** which may feature variables. It is part of a **Fragment** and represents where the **Fragment** will be woven, *i.e.* what the **Weaving** mechanism will match in the base process to weave the **Fragment**.
→ Pages 54, 55, 59, 60, 63–66, 68–70, 72, 77, 78, 80, 110, 130, 135, 145, 188

Pegasus: Open-source scientific workflow framework developed mainly at the University of Southern California, USA. It is focused on enactment over a variety of DCIs and many higher-level systems are based upon it, including WINGS.
→ [Deelman 05]
→ http://pegasus.isi.edu
→ Pages 155, 156, 159, 160, 187, 199

Petri net: A bipartite graph modeling a distributed system, whose nodes represent transitions and conditions in turns. Edges associate transitions with their pre-conditions and post-conditions.
→ Pages 12, 17, 107, 150, 155, 189

PIM (Platform-Independent Model): Class of models defined by the MDA and based on the abstraction level: platform-independent models are loosely-coupled with the infrastructure they run on, but algorithmically defined and thus somewhat inflexible.
→ Pages 5, 25, 26, 28, 29, 183, 191

Port: Element of the **Conceptual Workflow Model** which models either an argument or a product of an executable artifact underlying the **Activity** the **Port** is associated to.
→ Section 3.2.1
→ Pages 49, 53, 86, 95, 96, 101, 138, 183, 188–190, 193, 194

PROV: Extensive standard published by the W3C comprising a model for provenance data and a series of documents mapping the model to existing meta-models or specifying it in standards such as XML and Web Ontology Language (OWL).
→ `http://www.w3.org/TR/prov-overview/`.
→ Pages 3

Provenance: Meta-data detailing how something was produced. In the field of scientific workflows, the term provenance generally refers to the collection of information during workflow execution in order to associate result data to the input and parameters that produced them, to ease maintenance, to enhance reproducibility or many other uses.
→ Pages 3, 6, 146, 158, 193, 194

PSM (Platform-Specific Model): Class of models defined by the MDA and based on the abstraction level: platform-specific models are tightly-coupled with the infrastructure they run on and, as a result, are difficult to reuse.
→ Pages 5, 25, 29, 186, 191

QoS (Quality of Service): The set of considerations qualifying web services or other network products, including, for instance, availability, reliability and response time.
→ Pages 10, 23, 186

Ranking: Ranking is the second part of the **Discovery** process. It consists comparing the **Annotations** of candidate **Fragments** retrieved through **Matching** to that of the selected **Conceptual Workflow** and rank them based on how well the two sets match.
→ Section 4.4.3
→ Pages 81, 82

RDF (Resource Description Framework): According to the W3C, "*a standard model for data interchange on the web*". It is most notably used as a basis for most Semantic Web technologies.
→ `http://www.w3.org/TR/rdf-primer/`
→ Pages 34–36, 38–40, 63, 106, 194, 196, 197

RDFS (RDF Schema): According to the W3C, "*a general-purpose language for representing information in the Web. [It] describes how to use RDF to describe RDF vocabularies.*" It is by far the most basic Resource Description Framework (RDF)-based ontology language and defines merely what is needed to define a taxonomy.
→ `http://www.w3.org/TR/rdf-schema/`
→ Pages 36, 41, 63

Requirement: One of the two possible **Roles** for an **Annotation**. It means the **Annotation** represents a goal to achieve or a criterion to satisfy for the element bearing it.
→ Section 3.3.2
→ Pages 52–55, 58, 63, 68, 74–78, 80–84, 86, 130, 133–135

Role: Defining feature of an **Annotation** that defines its role inside a **Conceptual Workflow**.
→ Section 3.3.2
→ Pages 51, 52, 54, 77, 183, 195, 196

ROP (Role-Oriented Programming): Software development paradigm capturing the variability of how objects interact based on context. It is an especially successful paradigm in multi-agent systems, since it fits their collaboration-based design fairly well.
→ Pages 19, 20

Scientific Workflow Framework: A workflow framework designed to handle simulations and dedicated to one or multiple scientific workflow model(s).
→ Pages 2, 3, 5, 6, 10–12, 15, 17, 22, 29, 43, 50, 57, 94, 102, 107, 144, 146, 147, 151–153, 155–159, 161, 183, 184, 188–191, 193, 197–199

Scientific Workflow Model: A workflow model meant to formalize simulations.
→ Pages 2–6, 8, 10–12, 15, 17, 18, 24, 43, 47, 102, 107, 122, 144, 145, 155, 156, 162, 186, 187, 189

Scientific Workflow: A workflow meant to formalize and/or enact a simulation.
→ Pages 2–6, 8, 10–15, 17, 18, 22–24, 26, 29, 43, 50, 57, 94, 102, 107, 111, 113, 114, 122, 129, 144–147, 151, 156, 158–162, 183, 184, 186–190, 192–195, 198

SCUFL: Data-driven scientific workflow language underlying Taverna.
→ [Oinn 04]
→ `http://dev.mygrid.org.uk/wiki/display/developer/SCUFL2`
→ Pages 99, 152, 154, 157

Semantic Annotation: In general, any annotation that defines a semantic concept or links part of a document with a semantic concept defined elsewhere. In the **Conceptual Workflow Model**, see **Annotation**.
→ Section 3.3
→ Pages 15, 43, 51, 84, 86, 111, 144, 146, 150, 195, 199

Semantic Web: Also known as Web 3.0 and "*web of data*", it is a movement led by the W3C to promote the use of **Semantic Annotations** on web pages.
→ Pages 34, 51, 67, 144, 151, 194

SHIWA: Research project aimed at creating a platform to interoperate different scientific workflows languages.
→ [Krefting 11]
→ `http://www.shiwa-workflow.eu`
→ Pages 5, 94, 102, 146, 155, 176, 190

SIMRI: Magnetic Resonance (MR) imaging simulator seminally included in the VIP platform.
→ [Benoit-Cattin 05]
→ `http://www.simri.org/`
→ Pages 114, 117, 196

SimuBloch: Magnetic Resonance (MR) imaging simulator included soon after launch in the VIP platform. It quickly became more popular among users of the platform than the other MR simulator, SIMRI [Benoit-Cattin 05].
→ [Cao 12]

→ Pages 114, 118

Simulation: Scientific experiment carried out entirely or partially via computers.
→ Pages 1–6, 10, 12, 14, 17, 23, 24, 29, 30, 43, 47, 52, 57, 86, 112, 115, 117, 120, 121, 129, 144–146, 183, 185, 190, 191, 195

Sindbad: X-ray Computed Tomography (CT) simulator seminally included in the VIP platform.
→ [Tabary 09]
→ Pages 114, 120

SoC (Separation of Concerns): Software design principle of ensuring that loosely-related aspects of a system are developed separately and/or easily untangled. For instance, decoupling log handling from the program whose activity is logged respects the principle, whereas indiscriminately mixing both aspects does not.
→ Pages 8, 11, 18, 19, 22–24, 154, 158

SOP (Subject-Oriented Programming): Software development paradigm discriminating between intrinsic and extrinsic fields and methods. An object is thus the composition of "*subjects*", *i.e.* sets of fields and methods relevant to specific applications.
→ Pages 19, 20

SORTEO: Positron Emission Tomography (PET) simulator seminally included in the VIP platform.
→ [Gangemi 02]
→ `http://sorteo.cermep.fr/`
→ Pages 114, 120

SPARQL (SPARQL Protocol and RDF Query Language): Standard defined by the W3C to query and manipulate knowledge graphs.
See also RDF.
→ `http://www.w3.org/TR/rdf-sparql-query/`
→ Pages 38, 40, 41, 62–68, 72, 80, 83, 88–91, 109, 145, 166–168

Specification: One of the two possible **Roles** for an **Annotation**. It means the **Annotation** represents a goal achieved or a criterion satisfied by the element bearing it.
→ Section 3.3.2
→ Pages 52–54, 58, 68, 74, 75, 78, 80, 88, 89, 91, 92, 124–126, 129, 130, 138

SPL (Software Product Line): Branch of Feature-Oriented Programming (FOP) inspired by industrial product lining and using feature models to specify the exact set of viable and desirable feature compositions.
→ Pages 21–23

Sub-workflow: workflow or **Conceptual Workflow** contained in another workflow or **Conceptual Function**, relatively called its parent workflow.

→ Section 3.1.1
→ Pages 24, 46, 48, 52, 69, 73, 75, 88, 99, 102, 130, 142, 155, 188, 189, 191, 193

Swift: Open-source scientific workflow framework mainly developed at the Argonne National Laboratory, US. It is a scripting language and uses Karajan as its enactor.
→ [Zhao 07]
→ `http://www.ci.uchicago.edu/swift/main/index.php`
→ Pages 156, 190

Taverna: Open-source scientific workflow framework mainly developed by the myGrid team[3] at the University of Manchester, UK. Dedicated at first to bio-informaticians and the plethora of resources available to them via web services, it has evolved into a multi-domain, multi-platform and general-purpose scientific workflow framework.
→ [Missier 10a]
→ `http://www.taverna.org.uk/`
→ Pages 7, 12, 50, 94, 99, 154, 156, 157, 170, 195

Taxonomy: Hierarchical classification of concepts. For instance, the biological classification of species (*e.g.* "*homo sapiens sapiens*" which is a sub-class of "*homo sapiens*", itself a sub-class of "*homo*" and so on).
→ Pages 32, 36, 37, 52, 78, 79, 111

Transformation Process: Semi-automated transformation from a computation-independent **Conceptual Workflow** to an executable abstract workflow. It is done in two steps: **Mapping**, then **Conversion**.
→ Chapter 4
→ Pages 5, 6, 8, 29, 41, 43, 48, 51, 52, 57, 109, 130, 144, 145, 186, 188, 190–192, 198

Triana: Open-source scientific workflow framework developed mainly at Cardiff University, UK. It is dedicated to data processing pipelines and features a pure data-driven model as well as a wealth of data analysis tools.
→ [Taylor 07b]
→ `http://www.trianacode.org/`
→ Pages 13, 155, 157, 158

Triple Store: Database storing subject-predicate-object triples. Generally relying on more traditional (*e.g.* relational, object-oriented) databases, triple stores are often optimized for triple handling and most often feature inference and query engines.
→ Pages 191

Turtle: Compact textual syntax for RDF meant for human production and consumption. Its name derives from "*Terse RDF Triple Language*".
→ `http://www.w3.org/TeamSubmission/turtle/`
→ Pages 34–36, 38, 39, 65, 166

Type: Defining feature of an **Annotation** that makes it the sub-class of a class defined in an external ontology.
→ Section 3.3.1
→ Pages 51, 52, 54, 78, 83, 130, 183

[3] myGrid: → `http://www.mygrid.org.uk/`

UML (Unified Modeling Language): Standard modeling language commonly used to create formal graphical representations of objected-oriented systems.
→ http://www.omg.org/spec/UML/Current
→ Pages 26, 28, 37, 41, 43, 46, 49, 147, 163, 184, 192

URI (Uniform Resource Identifier): Standardized name, location or combination thereof identifying a resource, online or local, so that it can be interacted with via protocols.
→ Pages 34, 35, 38, 40, 63, 67, 68, 179, 180

VIP (Virtual Imaging Platform): French national research project whose goal was the integration of multiple modalities and organ models into a cohesive medical image simulation platform [Marion 11].
→ http://www.creatis.insa-lyon.fr/vip/
→ Pages 18, 52, 78, 94, 109, 111, 113–115, 117, 118, 120–122, 129–132, 139, 142, 144, 145, 179, 188, 195, 196

VisTrails: Open-source scientific workflow framework initially developed at the University of Utah, USA and now mainly developed at the University of New York, USA. It focuses on provenance of not only data but also scientific workflows themselves.
→ [Callahan 05]
→ http://www.vistrails.org
→ Pages 158, 159

W3C (World Wide Web Consortium): is "*an international community that develops open standards to ensure the long-term growth of the Web*".
→ http://www.w3.org/
→ Pages 3, 32, 34, 38, 94, 95, 193–196, 198, 199

Weaving: Weaving is a mechanism used in the **Mapping** step of the **Transformation Process** to merge **Fragments** into a **Conceptual Workflow**.
→ Section 4.2
→ Pages 8, 24, 41, 58, 59, 61–63, 66–73, 76, 77, 82, 91, 110, 130, 131, 133–135, 137, 145, 167, 184, 188, 192, 193

Web Service: According to the W3C: "*a software system designed to support interoperable machine-to-machine interaction over a network*". In other words, it is an Application Programming Interface (API) that can be accessed through web protocols.
→ Pages 2, 10–12, 14–16, 29, 40, 43, 146, 151, 152, 154, 156, 157, 162, 184, 194

Web Technologies: The set of technologies underlying and enabling internet, *e.g.* transfer protocols like TCP/IP, markup languages like XHTML, service protocols like SOAP.
→ Pages 2, 4

WfMC (Workflow Management Coalition): is "*a global organization of adopters, developers, consultants, analysts, as well as university and research groups engaged in workflow and [Business Process Management]*".
→ http://www.wfmc.org/
→ Pages 2, 199

WINGS: Open-source scientific workflow framework developed mainly at the University of Southern California, USA. It is based on Pegasus, defined at the **Conceptual Level** and leverages **Semantic Annotations** to assist design.
→ [Gil 11b]
→ http://www.wings-workflows.org
→ Pages 17, 146, 155, 159–161, 193

Workflow Framework: Set of tools to enable the use of workflows, most notably their creation, edition, enactment, deployment and monitoring.
→ Pages 23, 184, 195

Workflow: According to the Workflow Management Coalition (WfMC): "*the computerized facilitation or automation of a business process, in whole or part*".
→ Pages 2, 3, 7, 8, 10, 11, 13, 17, 18, 22–24, 49, 55, 59–70, 72, 73, 79, 84, 86, 87, 91, 99, 102, 107, 123, 124, 126, 129, 132–135, 137–139, 147, 149–162, 183–185, 187, 189, 191, 193, 195, 196, 199

WS-PGRADE: Open-source (APACHE licensed) scientific workflow framework mainly developed at the Laboratory of Parallel and Distributed Systems, MTA-SZTAKI, Hungary. It is part of the Graphical User Interface (GUI)/portal layer of the grid and cloud User Support Environment (gUSE) DCI gateway.
→ [Kacsuk 12]
→ http://www.p-grade.hu/
→ Pages 146, 155, 161, 162

XML (eXtensible Markup Language): Set of W3C recommendations that define rules and best practices for creating standard markup file formats.
→ http://www.w3.org/standards/xml/
→ Pages 28, 34, 36, 94, 95, 99, 147, 151, 155, 183, 187, 189, 192, 194, 199

XSLT (eXtensible Stylesheet Language Transformations): transformation language specified by the W3C to convert from an XML-based language to another.
→ http://www.w3.org/TR/xslt
→ Pages 28

BIBLIOGRAPHY

[Acher 12a] M. Acher, P. Collet, A. Gaignard, P. Lahire, J. Montagnat & R. France. *Composing Multiple Variability Artifacts to Assemble Coherent Workflows*. Software Quality Journal Special issue on Quality Engineering for Software Product Lines, vol. 20, no. 3-4, pages 689–734, September 2012.

[Acher 12b] M. Acher, P. Collet, P. Lahire & R. France. *FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models*. Science of Computer Programming (SCP) Special issue on programming languages (to appear), vol. 78, no. 6, pages 657–681, 2012.

[Akram 06] A. Akram, D. Meredith & R. Allan. *Evaluation of BPEL to Scientific Workflows*. In CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, pages 269–274, Washington, DC, USA, 2006. IEEE Computer Society.

[Alameda 07] J. Alameda, M. Christie, G. Fox, J. Futrelle, D. Gannon, M. Hategan, G. Kandaswamy, G. von Laszewski, M.A. Nacar, M. Pierce, E. Roberts, C. Severance & M. Thomas. *The Open Grid Computing Environments collaboration: portlets and services for science gateways*. Concurrency and Computation: Practice & Experience (CCPE), vol. 19, no. 6, pages 921–942, 2007.

[Aldrich 00] J. Aldrich. *Challenge Problems for Separation of Concerns*. In Proceedings, OOPSLA 2000 Workshop on Advanced Separation of Concerns(OOPSLA 2000), Minneapolis, USA, 2000.

[Alt 05] M. Alt & A. Hoheisel. *A grid workflow language using high-level petri nets*. In 6th international conference on Parallel processing and applied mathematics(PPAM'05), page 715722, Poznan, Poland, September 2005.

[Altintas 05] I. Altintas, A. Birnbaum, K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, Y. Potier & B. Ludäscher. *A Framework for the Design and Reuse of Grid Workflows*. In Scientific Applications of Grid Computing(LNCS), pages 295–299. Springer, 2005.

[Bachmann 10] A. Bachmann, M. Kunde, M. Litz & A. Schreiber. *Advances in generalization and decoupling of software parts in a scientific simulation workflow system*. In The Fourth International Conference on Advanced Engineering Computing and Applications in Sciences(ADVCOMP 2010), pages 34–38, 2010.

[Balligand 04] F. Balligand & V. Monfort. *A concrete solution for web services adaptability using policies and aspects*. In Proceedings of the 2nd international conference on Service oriented computing, ICSOC '04, pages 134–142, New York, NY, USA, 2004.

[Barga 07] R. Barga & D. Gannon. *Scientific versus Business Workflows*. In Workflows for e-Science [Taylor 07a], chapitre 2, pages 9–16.

[Barker 08] A. Barker & J. Van Hemert. *Scientific workflow: a survey and research directions*. In Proceedings of the 7th international conference on Parallel processing and applied mathematics(PPAM), PPAM'O7, pages 746–753, Berlin, Heidelberg, Germany, 2008. Springer-Verlag.

[Batory 92] D. Batory & S. O'Malley. *The design and implementation of hierarchical software systems with reusable components*. ACM Trans. Softw. Eng. Methodol., vol. 1, no. 4, pages 355–398, October 1992.

[Batory 03] D. Batory, J.N. Sarvela & A. Rauschmayer. *Scaling step-wise refinement*. In 25th International Conference on Software Engineering (ICSE '03), pages 187–197, Portland, Oregon, 2003. IEEE Computer Society.

[Batory 04] D. Batory. *Feature-Oriented Programming and the AHEAD Tool Suite*. In Proceedings of the 26th International Conference on Software Engineering(ICSE'04), pages 702–703, Washington, DC, USA, 2004. IEEE Computer Society.

[Bechhofer 10] S. Bechhofer, D. de Roure, M. Gamble, C. Goble & I. Buchan. *Research Objects: Towards Exchange and Reuse of Digital Knowledge*. The Future of the Web for Collaborative Science (FWCS), April 2010.

[Becht 99] M. Becht, T. Gurzki, J. Klarmann & M. Muscholl. *ROPE: role oriented programming environment for multiagent systems*. In Cooperative Information Systems, 1999. CoopIS'99. Proceedings. 1999 IFCIS International Conference on(CoopIS), pages 325–333. IEEE, 1999.

[Belakhdar 96] O. Belakhdar & J. Ayel. *Modelling approach and tool for designing protocols for automated cooperation in multi-agent systems*. In Agents Breaking Away(LNCS), pages 100–115. Springer Berlin Heidelberg, 1996.

[Belhajjame 12] K. Belhajjame, O. Corcho, D. Garijo, J. Zhao, P. Missier, D. Newman, R. Palma, S. Bechhofer, E. Garcia-Cuesta, J.M. Gómez-Pérez, G. Klyne, K. Page, M. Roos, J.E. Ruiz, S. Soiland-Reyes, L. Verdes-Montenegro, D. de Roure & C. Goble. *Workflow-Centric Research Objects: A First-Class Citizen in the Scholarly Discourse*. In ESWC2012 Workshop on the Future of Scholarly Communication in the Semantic Web(SePublica2012), pages 1–12, Heraklion, Greece, May 2012.

[Benoit-Cattin 03] H. Benoit-Cattin, F. Bellet, J. Montagnat & C. Odet. *Magnetic Resonance Imaging (MRI) Simulation on a Grid Computing Architecture*. In Biogrid'03, proceedings of the IEEE CCGrid03(Biogrid'03), pages 582–587, Tokyo, Japan, May 2003.

[Benoit-Cattin 05] H. Benoit-Cattin, G. Collewet, B. Belaroussi, H. Saint-Jalmes & C. Odet. *The SIMRI project : a versatile and interactive MRI simulator*. Journal of Magnetic Resonance Imaging (JMRI), vol. 173, no. 1, pages 97–115, March 2005.

[Berthold 07] M.R. Berthold, N. Cebron, F. Dill, T.R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel & B. Wiswedel. *KNIME: The Konstanz Information Miner*. In GfKl(GfKl), Studies in Classific, pages 319–326, Freiburg, Germany, March 2007. Springer.

[Bowers 05] S. Bowers & B. Ludäscher. *Actor-Oriented Design of Scientific Workflows*. In 24th Intl. Conf. on Conceptual Modeling(ER'05), LNCS 3716, pages 369–384. Springer, 2005.

[Bowers 06] S. Bowers, B. Ludäscher, A.H.H. Ngu & T. Critchlow. *Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow*. In IEEE Workshop on Workflow and Data Flow for Scientific Applications(SciFlow), Atlanta, USA, April 2006.

[Cabri 04] G. Cabri, L. Ferrari & L. Leonardi. *Agent role-based collaboration and coordination: a survey about existing approaches*. In Systems, Man and Cybernetics, 2004 IEEE International Conference on, volume 6, pages 5473–5478. IEEE, 2004.

[Callahan 05] S.P. Callahan, P.J. Crossno, J. Freire, C.E. Scheidegger, C.T. Silva & H.T. Vo. *VisTrails: enabling interactive multiple-view visualizations*. In Visualization, 2005. VIS 05. IEEE, pages 135–142. IEEE, 2005.

[Cannataro 07] M. Cannataro, P. H. Guzzi, T. Mazza, G. Tradigo & P. Veltri. *Using ontologies for preprocessing and mining spectra data on the Grid*. Future Generation Computer Systems (FGCS), vol. 23, no. 1, pages 55–60, 2007.

[Cao 12] F. Cao, O. Commowick, E. Bannier, J.-C. Ferré, G. Edan & C. Barillot. *MRI Estimation of T1 Relaxation Time Using a Constrained Optimization Algorithm*. In Multimodal Brain Image Analysis(LNCS), pages 203–214. Springer Berlin Heidelberg, 2012.

[Carroll 04] J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne & K. Wilkinson. *Jena: implementing the semantic web recommendations*. In Proceedings of the 13th international World Wide Web conference on, WWW Alt. '04, pages 74–83, New York, NY, US, 2004. ACM.

[Cerezo 11] N. Cerezo & J. Montagnat. *Scientific Workflow Reuse through Conceptual Workflows*. In Proceedings of the 6th Workshop on Workflows in Support of Large-Scale Science, WORKS'11, pages 1–10, Seattle, WA, USA, November 2011. ACM.

[Cerezo 13] N. Cerezo, J. Montagnat & M. Blay-Fornarino. *Computer-Assisted Scientific Workflow Design*. Journal of Grid Computing (JOGC), vol. 11, no. 3, pages 585–610, September 2013.

[Charfi 04] A. Charfi & M. Mezini. *Aspect-Oriented Web Service Composition with AO4BPEL.* In European Conference On Web Services(ECOW), volume 3250, page 168, Erfurt, Germany, September 2004. Springer Berlin / Heidelberg.

[Chen 76] P.P.-S. Chen. *The entity-relationship modeltoward a unified view of data.* ACM Transactions on Database Systems (TODS), vol. 1, no. 1, pages 9–36, 1976.

[Chin 11] G. Chin, C. Sivaramakrishnan, T. Critchlow, K. Schuchardt & A.H.H. Ngu. *Scientist-Centered Workflow Abstractions via Generic Actors, Workflow Templates, and Context-Awareness for Groundwater Modeling and Analysis*. In IEEE World Congress on Services(SERVICES), pages 176–183, July 2011.

[Clements 01] P. Clements & L. Northrop. Software Product Lines: Practices and Patterns. Addison Wesley, Boston, USA, 2001.

[Coady 01] Y. Coady, G. Kiczales, M. Feeley & G. Smolyn. *Using aspectC to improve the modularity of path-specific customization in operating system code*. SIGSOFT Softw. Eng. Notes, vol. 26, no. 5, pages 88–98, September 2001.

[Corby 05] O. Corby, R. Dieng, C. Faron-Zucker & F. Gandon. *Ontology-based Approximate Query Processing for Searching the Semantic Web with CORESE.* Technical Report RR-5621, INRIA, Sophia Antipolis, France, July 2005.

[Courbis 05] C. Courbis & A. Finkelstein. *Towards aspect weaving applications*. In Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on(ICSE), pages 69–77, 2005.

[Curcin 08] V. Curcin & M. Ghanem. *Scientific workflow systems-can one size fit all?* In Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International, pages 1–9, Cairo, Egypt, December 2008. IEEE.

[Czarnecki 03] K. Czarnecki & S. Helsen. *Classification of Model Transformation Approaches*. In 2nd OOPSLA03 Workshop on Generative Techniques in the Context of MDA(OOPSLA'03), Anaheim (Califormie), October 2003. ACM.

[De Roo 08] A.J. De Roo, M.F.H. Hendricks, W.K. Havinga, P.E.A. Durr & L.M.J. Bergmans. *Compose*: a Language- and Platform-Independent Aspect Compiler for Composition Filters*. In First International Workshop on Advanced Software Development Tools and Techniques(WASDeTT 2008), pages 1–14, Paphos, Cyprus, July 2008.

[Deelman 05] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob & D.S. Katz. *Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems*. Scientific Programming Journal, vol. 13, no. 3, pages 219–237, 2005.

[Deelman 09] E. Deelman, D. Gannon, M. Shields & I. Taylor. *Workflows and e-Science: An overview of workflow system features and capabilities*. Future Generation Computer Systems (FGCS), vol. 25, no. 5, pages 528–540, May 2009.

[Demsky 02] B. Demsky & M. Rinard. *Role-based exploration of object-oriented programs*. In Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on(ICSE), pages 313–324. IEEE, 2002.

[Dijkstra 82] E.W. Dijkstra. *On the Role of Scientific Thought*. In Selected Writings on Computing: A personal Perspective, pages 60–66. Springer New York, 1982.

[Emmerich 05] W. Emmerich, B. Butchart, L. Chen, B. Wassermann & S. Price. *Grid Service Orchestration Using the Business Process Execution Language (BPEL)*. Journal of Grid Computing (JOGC), vol. 3, no. 3-4, pages 283 – 304, September 2005.

[Fahringer 05] T. Fahringer, J. Qin & S. Hainzer. *Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language*. In fifth IEEE International Symposium on Cluster Computing and the Grid(CCGrid'05), volume 2, pages 676–685, Cardiff, UK, May 2005. IEEE Computer Society.

[Fahringer 07] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon & M. Wieczorek. *ASKALON: a development and grid computing environment for scientific workflows*. In Workflows for e-Science [Taylor 07a], chapitre 27, pages 450–471.

[Feigenbaum 83] E.A. Feigenbaum & P. McCorduck. The fifth generation : artificial intelligence and Japan's computer challenge to the world. Addison-Wesley, first edition, 1983.

[Fellbaum 10] C. Fellbaum. *WordNet*. In Theory and Applications of Ontology: Computer Applications, pages 231–243. Springer Netherlands, 2010.

[Fensel 01] D. Fensel, F. van Harmelen, I. Horrocks, D.L. McGuinness & P.F. Patel-Schneider. *OIL: an ontology infrastructure for the Semantic Web*. IEEE Intelligent Systems, vol. 16, no. 2, pages 38–45, 2001.

[Forestier 11] G. Forestier, A. Marion, H. Benoit-Cattin, P. Clarysse, D. Friboulet, T. Glatard, P. Hugonnard, C. Lartizien, H. Liebgott, J. Tabary & B. Gibaud. *Sharing object models for multi-modality medical image simulation: A semantic approach*. In 24th International Symposium on Computer-Based Medical Systems(CBMS), pages 1–6. IEEE, June 2011.

[Gaignard 13] A. Gaignard. *Distributed knowledge sharing and production through collaborative e-Science platforms*. PhD thesis, Université de Nice Sophia Antipolis, France, March 2013.

[Gamma 93] E. Gamma, R. Helm, R. Johnson & J. Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Medical Image Analysis (MedIA), vol. 707, pages 406–431, 1993.

[Gangemi 02] A. Gangemi, N. Guarino, C. Masolo, A. Oltramari & L. Schneider. *Sweetening Ontologies with DOLCE*. In Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web(LNCS), pages 166–181. Springer Berlin Heidelberg, 2002.

[García-Silva 12] A. García-Silva, O. Corcho, H. Alani & A. Gómez-Pérez. *Review of the state of the art: Discovering and Associating Semantics to Tags in Folksonomies*. The Knowledge Engineering Review, vol. 27, no. 1, pages 57–85, March 2012.

[Garijo 11] D. Garijo & Y. Gil. *A new approach for publishing workflows: abstractions, standards, and linked data*. In Proceedings of the 6th workshop on Workflows in support of large-scale science(WORKS), pages 47–56, New York, NY, USA, 2011. ACM.

[Garijo 13] D. Garijo, O. Corcho & Y. Gil. *Detecting common scientific workflow fragments using templates and execution provenance*. In Proceedings of the seventh international conference on Knowledge capture(K-CAP '13), pages 33–40, Banff, Canada, 2013. ACM.

[Gaspar 10] W. Gaspar, L. Machado da Silva, R.M.M. Braga & F. Campos. *SW-Ontology - A Proposal for Semantic Modeling of a Scientific Workflow Management System*. In Proceedings of the 12th International Conference on Enterprise Information Systems(ICEIS 2010), volume 1, pages 115–120, Madeira, Portugal, June 2010. SciTePress.

[Georgakopoulos 95] D. Georgakopoulos, M. Hornick & A. Sheth. *An overview of workflow management: From process modeling to workflow automation infrastructure*. Distributed and Parallel Databases, vol. 3, no. 2, pages 119–153, 1995.

[Gibaud 11] B. Gibaud, G. Kassel, M. Dojat, B. Batrancourt, F. Michel, A. Gaignard & J. Montagnat. *NeuroLOG: sharing neuroimaging data using an ontology-based federated approach*. In American Medical Informatics Association(AMIA'2011), volume 2011, page 472480, Washington DC, USA, October 2011. AMIA.

[Gil 07] Y. Gil, E. Deelman, M.H. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau & J. Myers. *Examining the Challenges of Scientific Workflows*. Computer, vol. 40, pages 24–32, 2007.

[Gil 11a] Y. Gil, J. Kim, P.A. Gonzales-Calero, J. Kim, J. Moody & V. Ratnakar. *A semantic framework for automatic generation of computational workflows using distributed data and component catalogues*. Journal of Experimental & Theoretical Artificial Intelligence, vol. 23, no. 4, pages 389–467, 2011.

[Gil 11b] Y. Gil, V. Ratnakar, K. Jihie, J. Moody, E. Deelman, P.A. Gonzales-Calero & P. Groth. *Wings: Intelligent Workflow-Based Design of Computational Experiments*. IEEE Intelligent System, vol. 26, no. 1, pages 62–72, January 2011.

[Glatard 08] T. Glatard, J. Montagnat, D. Lingrand & X. Pennec. *Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR*. International Journal of High Performance Computing Applications (IJHPCA) Special issue on Special Issue on Workflows Systems in Grid Environments, vol. 22, no. 3, pages 347–360, August 2008.

[Glatard 13] T. Glatard, C. Lartizien, B. Gibaud, R. Ferreira Da Silva, G. Forestier, F. Cervenansky, M. Alessandrini, H. Benoit-Cattin, O. Bernard, S. Camarasu-Pop, N. Cerezo, P. Clarysse, A. Gaignard, P. Hugonnard, H. Liebgott, S. Marache, A. Marion, J. Montagnat, J. Tabary & D. Friboulet. *A Virtual Imaging Platform for multi-modality medical image simulation*. IEEE Transactions on Medical Imaging (TMI), vol. 32, no. 1, pages 110 – 118, January 2013.

[Goderis 05] A. Goderis, U. Sattler, P. Lord & C. Goble. *Seven Bottlenecks to Workflow Reuse and Repurposing*. In The Semantic Web ISWC 2005(LNCS), pages 323–337. Springer, Heidelberg, Germany, 2005.

[Goecks 10] J. Goecks, A. Nekrutenko & J. Taylor. *Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences*. Genome Biology, vol. 11, no. 8, page R86, 2010.

[Görlach 11] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann & M. Reiter. *Conventional Workflow Technology for Scientific Simulation*. In Guide to e-Science, Computer Communications and Networks, pages 323–352. Springer London, 2011.

[Haase 10] P. Haase, T. Mathäss & M. Ziller. *An evaluation of approaches to federated query processing over linked data*. In Proceedings of the 6th International Conference on Semantic Systems(I-SEMANTICS '10), pages 5:1–5:9, Graz, Austria, 2010. ACM.

[Harrison 93] W. Harrison & H. Ossher. *Subject-oriented programming: a critique of pure objects*. In Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications(OOPSLA '93), pages 411–428, New York, NY, USA, 1993. ACM.

[Hartig 10] O. Hartig, J. Sequeda, J. Taylor & P. Sinclair. *How to consume linked data on the web: tutorial description*. In Proceedings of the 19th international conference on World wide web(WWW '10), pages 1347–1348, Raleigh, North Carolina, USA, 2010. ACM.

[Hauder 11] M. Hauder, Y. Gil, Y. Liu, R. Sethi & H. Jo. *Making data analysis expertise broadly accessible through workflows*. In Proceedings of the 6th workshop on Workflows in support of large-scale science(WORKS), pages 77–86, New York, NY, USA, 2011. ACM.

[Hirschfeld 03] R. Hirschfeld. *AspectS - Aspect-Oriented Programming with Squeak*. In Objects, Components, Architectures, Services, and Applications for a Networked Worl(LNCS), pages 216–232. Springer Berlin Heidelberg, 2003.

[Hürsch 95] W.L. Hürsch & C.V. Lopes. *Separation of Concerns*. Technical report, Computer Science Dept., Northeastern University, Boston, USA, 1995.

[Jensen 04] J.A. Jensen. *Simulation of advanced ultrasound systems using Field II*. In IEEE International Symposium on Biomedial Imaging: Nano to Macro, pages 636–639, April 2004.

[Kacsuk 12] P. Kacsuk, Z. Farkas, M. Kozlovszky, G. Hermann, A. Balasko, K. Karoczkai & I. Marton. *WS-PGRADE/gUSE Generic DCI Gateway Framework for a Large Variety of User Communities*. Journal of Grid Computing (JOGC), vol. 10, no. 4, pages 601–630, 2012.

[Kassel 10] G. Kassel. *A formal ontology of artefacts*. Applied Ontology, vol. 5, no. 3, pages 223–246, 2010.

[Kendall 99] E.A. Kendall. *Role modelling for agent system analysis, design, and implementation*. In Agent Systems and Applications, 1999 and Third International Symposium on Mobile Agents. Proceedings. First International Symposium on, pages 204–218, 1999.

[Kent 02] S. Kent. *Model Driven Engineering*. In Integrated Formal Method(LNCS), pages 286–298. Springer Berlin Heidelberg, 2002.

[Kiczales 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier & J. Irwin. *Aspect-Oriented Programming*. In Proceedings of the European Conference on Object-Oriented Programming, volume 1241, pages 220–242, 1997.

[Kiczales 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W.G. Griswold. *An overview of AspectJ*. In European Conference on Object-Oriented Programming(ECOOP'01), pages 327–353. Springer-Verlag, 2001.

[Kifer 95] M. Kifer, G. Lausen & J. Wu. *Logical foundations of object-oriented and frame-based languages*. Journal of the ACM (JACM), vol. 42, no. 4, pages 741–843, July 1995.

[Krefting 11] D. Krefting, T. Glatard, V. Korkhov, J. Montagnat & S. Olabarriaga. *Enabling Grid Interoperability at Workflow Level*. In Grid Workflow Workshop 2011(GWW'11), Köln, Germany, March 2011.

[Kristensen 96] B.B. Kristensen & K. sterbye. *Roles: conceptual abstraction theory and practical language issues*. Theory and Practice of Object Systems, vol. 2, no. 3, pages 143–160, 1996.

[Kusiak 97] A. Kusiak, T. Letsche & A. Zakarian. *Data modelling with IDEF1x*. International Journal of Computer Integrated Manufacturing (IJCIM), vol. 10, no. 6, pages 470–486, 1997.

[Lando 07] P. Lando, A. Lapujade, G. Kassel & F. Fürst. *Towards a general ontology of computer programs*. In International Conference on Software and Data Technologies(ICSOFT'07), pages 25–27, Barcelona, Spain, July 2007.

[Lee 00] E.A. Lee & S. Neuendorffer. *MoML - A Modeling Markup Language in XML, Version 0.4*. Technical Report UCB/ERL M00/12, University of California, Berkeley, CA 94720, March 2000.

[Lenat 91] D.B. Lenat & R.V. Guha. *The evolution of CycL, the Cyc representation language*. ACM SIGART Bulletin (SIGART), vol. 2, pages 84–87, June 1991.

[Ludäscher 06] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao & Y. Zhao. *Scientific Workflow Management and the Kepler System*. Concurrency and Computation: Practice & Experience (CCPE), vol. 18, no. 10, pages 1039 – 1065, August 2006.

[Maheshwari 10] K. Maheshwari & J. Montagnat. *Scientific workflows development using both visual-programming and scripted representations*. In International Workshop on Scientific Workflows(SWF'10), Miami, Florida, USA, July 2010. IEEE.

[Malik 12] M.J. Malik, T. Fahringer & R. Prodan. *Semi-automatic Composition of Ontologies for ASKALON Grid Workflows*. In Euro-Par 2011: Parallel Processing Workshops(LNCS), volume 7155 of *Lecture Notes in Com*, pages 169–180. Springer Berlin Heidelberg, 2012.

[Marion 11] A. Marion, G. Forestier, H. Liebgott, C. Lartizien, H. Benoit-Cattin, S. Camarasu-Pop, T. Glatard, R. Ferreira Da Silva, P. Clarysse, S. Valette, B. Gibaud, P. Hugonnard, J. Tabary & D. Friboulet. *Multi-modality image simulation of biological models within VIP*. In 24th International Symposium on Computer-Based Medical Systems(CBMS), pages 1–6, Bristol, UK, June 2011.

[Masolo 03] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari & L. Schneider. *The WonderWeb Library of Foundational Ontologies and the DOLCE ontology. WonderWeb Deliverable D18, Final Report*. Technical report, ISTC-CNR, Padova, Italy, December 2003.

[Matheus 06] C.J. Matheus, K. Baclawski & M.M. Kokar. *BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rule*. In Rules and Rule Markup Languages for the Semantic Web, Second International Conference on, pages 67–74. IEEE, 2006.

[McGuinness 02] D.L. McGuinness, R. Fikes, J. Hendler & L.A. Stein. *DAML+OIL: an ontology language for the Semantic Web*. IEEE Intelligent Systems, vol. 17, no. 5, pages 72–80, 2002.

[McPhillips 09] T.M McPhillips, S. Bowers, D. Zinn & B. Ludäscher. *Scientific workflow design for mere mortals*. Future Generation Computer Systems (FGCS), vol. 25, no. 5, pages 541–551, 2009.

[Miller 03] J. Miller & J. Mukerji. *MDA Guide Version 1.0.1*. Technical report, OMG, 2003.

[Missier 10a] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn & C. Goble. *Taverna, reloaded*. In SSDBM 2010, Heidelberg, Germany, June 2010.

[Missier 10b] P. Missier, K. Wolstencroft, F. Tanoh, S. Bechhofer, K. Belhajjame, S. Pettifer & C. Goble. *Functional Units: Abstractions for Web Service Annotations*. In 6th World Congress on Services(SERVICES), pages 306–313, July 2010.

[Montagnat 09] J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari & M. Blay-Fornarino. *A data-driven workflow language for grids based on array programming principles*. In Workshop on Workflows in Support of Large-Scale Science(WORKS'09), pages 1–10, Portland, USA, November 2009. ACM.

[Mosser 10] S. Mosser, M. Blay-Fornarino & R. France. *Workflow Design using Fragment Composition (Crisis Management System Design through ADORE)*. Transactions on Aspect-Oriented Software Development (TAOSD) Special issue on Aspect Oriented Modeling, pages 1–34, 2010.

[Mosser 12] S. Mosser & M. Blay-Fornarino. *ADORE, a Logical Meta-model Supporting Business Process Evolution*. Science of Computer Programming (SCP) IF=1.282, pages 1–35, 2012.

[Motik 09] B. Motik, R. Shearer & I. Horrocks. *Hypertableau Reasoning for Description Logics*. Journal of Artificial Intelligence Research (JAIR), vol. 36, pages 165–228, 2009.

[Neubauer 05] F. Neubauer, A. Hoheisel & J. Geiler. *Workflow-based Grid applications*. Future Generation Computer Systems (FGCS), vol. 22, no. 1-6, pages 6–15, September 2005.

[Ogasawara 09] E. Ogasawara, C. Paulino, L. Murta, C. Werner & M. Mattoso. *Experiment Line: Software Reuse in Scientific Workflow*. In Scientific and Statistical Database Management(LNCS), pages 264–272. Springer Berlin Heidelberg, 2009.

[Oinn 04] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Greenwood, C. Goble, A. Wipat, P. Li & T. Carver. *Delivering web service coordination capability to users*. In Proceedings of the 13th international World Wide Web conference on Alternate track papers and posters(WWW Alt.), pages 438–439, New York, NY, USA, 2004. ACM.

[Ortiz 06] G. Ortiz & F. Leymann. *Combining WS-Policy and Aspect-Oriented Programming*. In Telecommunications, 2006. International Conference on Internet and Web Applications and Services/Advanced International Conference on, AICT-ICIW '06, pages 143–149. IEEE, 2006.

[Pawlak 01] R. Pawlak, L. Duchien, G. Florin & L. Seinturier. *JAC: A Flexible Solution for Aspect-Oriented Programming in Java*. In Metalevel Architectures and Separation of Crosscutting Concerns(LNCS), pages 1–24. Springer Berlin Heidelberg, 2001.

[Pessemier 08] N. Pessemier, L. Seinturier & L. Duchien. *A component-based and aspect-oriented model for software evolution*. International Journal of Computer Applications in Technology, vol. 1/2, no. 31, pages 94–105, 2008.

[Plankensteiner 11] K. Plankensteiner, J. Montagnat & R. Prodan. *IWIR: A Language Enabling Portability Across Grid Workflow Systems*. In Workshop on Workflows in Support of Large-Scale Science(WORKS'11), Seattle, USA, November 2011.

[Prehofer 93] C. Prehofer. *Feature-oriented programming: A fresh look at objects*. In ECOOP'97 Object-Oriented Programming(LNCS), pages 419–443. Springer Berlin / Heidelberg, 1993.

[Qin 08] J. Qin & T. Fahringer. *A novel domain oriented approach for scientific grid workflow composition*. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, Austin, TX, USA, 2008. IEEE Press.

[Reenskaug 96] T. Reenskaug, P. Wold & O.A. Lehne. Working with objects: the OOram software engineering method. Manning Greenwich, 1996.

[Reese 06] C. Reese, J. Ortmann, S. Offermann, D. Moldt, K. Markwardt & T. Carl. *Fragmented Workflows Supported by an Agent Based Architecture*. In Agent-Oriented Information Systems III(LNCS), pages 200–215. Springer Berlin Heidelberg, 2006.

[Reilhac 04] A. Reilhac, C. Lartizien, N. Costes, S. Sans, C. Comtat, Roger N. Gunn & Alan C. Evans. *PET-SORTEO: a Monte Carlo-based Simulator with high count rate capabilities*. IEEE Transactions on Nuclear Science (TNS), vol. 51, no. 1, pages 46–52, February 2004.

[Sadiq 01] S. Sadiq, W. Sadiq & M. Orlowska. *Pockets of Flexibility in Workflow Specification*. In Conceptual Modeling ER 2001(LNCS), pages 513–526. Springer Berlin Heidelberg, 2001.

[Schumm 11] D. Schumm, D. Karastoyanova, O. Kopp, F. Leymann, M. Sonntag & S. Strauch. *Process fragment libraries for easier and faster development of process-based applications*. Journal of Systems Integration, vol. 2, no. 1, pages 39–55, 2011.

[Schumm 12] D. Schumm, D. Dentsas, M. Hahn, D. Karastoyanova, F. Leymann & M. Sonntag. *Web Service Composition Reuse through Shared Process Fragment Libraries*. In Web Engineering(ICWE), volume 7387 of *LNCS*, pages 498–501, Berlin, Germany, 2012. Springer Berlin Heidelberg.

[Seinturier 09] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni & J.-B. Stefani. *Reconfigurable SCA Applications with the FraSCAti Platform*. In Services Computing, 2009. IEEE International Conference on, SCC '09, pages 268–275, 2009.

[Shepitsen 08] A. Shepitsen, J. Gemmell, B. Mobasher & R. Burke. *Personalized recommendation in social tagging systems using hierarchical clustering*. In Proceedings of the 2008 ACM conference on Recommender systems(RecSys '08), pages 259–266, Lausanne, Switzerland, 2008. ACM.

[Shields 07] M. Shields. *Control Versus Data-Driven Workflows*. In Workflows for e-Science [Taylor 07a], chapitre 11, pages 167 – 173.

[Silva 08] A. P. C. Silva, V. C. M. Borges & M. A. R. Dantas. *A framework for mobile grid environments based on semantic integration of ontologies and workflow-based applications*. Infocomp Journal of Computer Science, vol. 7, no. 1, pages 59–66, 2008.

[Singh 10] Y. Singh & M. Sood. *The Impact of the Computational Independent Model for Enterprise Information System Development*. International Journal of Computer Applications, vol. 11, no. 8, pages 24–28, 2010.

[Sirin 07] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur & Y. Katz. *Pellet: A practical OWL-DL reasoner*. Web Semantics: Science, Services and Agents on the World Wide Web (JOWS), vol. 5, no. 2, pages 51–53, 2007.

[Slominski 07] A. Slominski. *Adapting BPEL to Scientific Workflows*. In Workflows for e-Science [Taylor 07a], chapitre 14, pages 208–226.

[Sonntag 10] M. Sonntag, D. Karastoyanova & F. Leymann. *The Missing Features of Workflow Systems for Scientific Computations*. In Proceedings of the 3rd Grid Workflow Workshop(GWW), pages 209–216, Paderborn, Germany, 2010. Gesellschaft für Informatik.

[Sonntag 13] M. Sonntag & D. Karastoyanova. *Model-as-you-go: An Approach for an Advanced Infrastructure for Scientific Workflows*. Journal of Grid Computing (JOGC), pages 1–31, 2013.

[Spinczyk 02] O. Spinczyk, A. Gal & W. Schröder-Preikschat. *AspectC++: an aspect-oriented extension to the C++ programming language*. In Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications(CRPIT'02), pages 53–60, Sydney, Australia, 2002. Australian Computer Society, Inc.

[Sroka 09] J. Sroka, J. Hidders, P. Missier & C. Goble. *A formal semantics for the Taverna 2 workflow model*. Journal of Computer and System Sciences (JCSS), vol. 76, no. 6, pages 490–508, November 2009.

[Tabary 09] J. Tabary, S. Marache, S. Valette, W.P. Segars & C. Lartizien. *Realistic X-Ray CT Simulation of the XCAT Phantom with SINDBAD*. In IEEE NSS and MIC Conference, Orlando, USA, October 2009.

[Taylor 07a] I. Taylor, E. Deelman, D. Gannon & M. Shields. Workflows for e-Science. Springer-Verlag, 2007.

[Taylor 07b] I. Taylor, M. Shields, I. Wang & A. Harrison. *The Triana Workflow Environment: Architecture and Applications*. In Workflows for e-Science [Taylor 07a], chapitre 20, pages 320–339.

[Temal 08] L. Temal, M. Dojat, G. Kassel & B. Gibaud. *Towards an ontology for sharing medical images and regions of interest in neuroimaging*. Bioinformatics journal, vol. 41, no. 5, pages 766–778, 2008.

[Trujillo 07] S. Trujillo, D. Batory & O. Diaz. *Feature Oriented Model Driven Development: A Case Study for Portlets*. In Proceedings of the 29th international conference on Software Engineering(ICSE'07), pages 44–53, Washington, DC, USA, 2007. IEEE Computer Society.

[Tsarkov 06] D. Tsarkov & I. Horrocks. *FaCT++ Description Logic Reasoner: System Description*. In Automated Reasoning (LNCS), pages 292–297. Springer Berlin Heidelberg, 2006.

[Van Der Aalst 03] W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski & A.P. Barros. *Workflow patterns*. Distributed and Parallel Databases, vol. 14, no. 3, pages 5–51, July 2003.

[von Laszewski 01] G. von Laszewski, I. Foster, J. Gawor & P. Lane. *A Java commodity grid kit*. Concurrency and Computation: Practice & Experience (CCPE), vol. 13, no. 8-9, pages 645–662, 2001.

[von Laszewski 07] G. von Laszewski, D. Kodeboyina & M. Hategan. *Java CoG Kit workflow*. In Workflows for e-Science [Taylor 07a], chapitre 21, pages 340–356.

[Wassermann 07] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen & J. Patel. *Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling*. In Workflows for e-Science [Taylor 07a], chapitre 26, pages 428–449.

[Withers 10] D. Withers, E. Kawas, L. McCarthy, B. Vandervalk & M. Wilkinson. *Semantically-Guided Workflow Construction in Taverna: The SADI and BioMoby Plug-Ins*. In Leveraging Applications of Formal Methods, Verification, and Validation(LNCS), pages 301–312. Springer Berlin Heidelberg, 2010.

[Wolstencroft 07] K. Wolstencroft, P. Alper, D. Hull, C. Wroe, P. Lord, R. Stevens & C. Goble. *The myGrid Ontology: Bioinformatics Service Discovery*. International Journal of Bioinformatics Research and Applications (IJBRA), 2007.

[Wroe 07] C. Wroe, C. Goble, A. Goderis, P. Lord, S. Miles, J. Papay, P. Alper & L. Moreau. *Recycling workflows and services through discovery and reuse*. Concurrency and Computation: Practice & Experience (CCPE), vol. 19, no. 2, pages 181–194, 2007.

[Yu 05] J. Yu & R. Buyya. *A taxonomy of scientific workflow systems for grid computing*. ACM SIGMOD records (SIGMOD), vol. 34, no. 3, pages 44–49, September 2005.

[Zhao 07] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun & M. Wilde. *Swift: Fast, Reliable, Loosely Coupled Parallel Computation*. In IEEE International Workshop on Scientific Workflows, Salt-Lake City, Utah, USA, July 2007.

[Zhao 08] Y. Zhao, I. Raicu & I. Foster. *Scientific Workflow Systems for 21st Century, New Bottle or New Wine?* In Services - Part I, 2008. IEEE Congress on, pages 467–471, Honolulu, HI, USA, July 2008. IEEE.

Abstract

Workflows are increasingly adopted to describe large-scale data- and compute-intensive scientific simulations which leverage the wealth of distributed data sources and computing infrastructures. Nonetheless, most scientific workflow formalisms remain difficult to exploit for scientists who are neither experts nor enthusiasts of distributed computing, because they mix the scientific processes they model with their implementations, blurring the lines between what is done and how it is done, as well as between what is and what is not infrastructure-dependent.

Our objective is to improve scientific workflow accessibility and ease scientific workflow design and reuse, by elevating the abstraction level, emphasizing the scientific experiment over technicalities, ensuring proper separation between functional and non-functional concerns and leveraging domain knowledge and know-how.

The main contributions of this work are: (i) a multi-level structurally flexible semantic scientific workflow model, called the Conceptual Workflow Model, which lets users design simulations at a computation-independent level and focus on domain goals and methods; and (ii) a computer-assisted Transformation Process relying on knowledge engineering technologies to help users transform their high-level simulation models into executable workflow artifacts which can be delegated to third-party frameworks for enactment.

Résumé

Les workflows sont de plus en plus souvent adoptés pour la modélisation de simulations scientifiques de grande échelle, aussi bien en matière de données que de calculs. Ils profitent de l'abondance de sources de données et infrastructures de calcul distribuées. Néanmoins, la plupart des formalismes de workflows scientifiques restent difficiles à exploiter pour des utilisateurs n'ayant pas une grande expertise de l'algorithmique distribuée, car ces formalismes mélangent les processus scientifiques qu'ils modélisent avec leurs implémentations. Ainsi, ils ne permettent pas de distinguer entre les objectifs et les méthodes, ni de repérer les particularités d'une implémentation ou de l'infrastructure sous-jacente.

Le but de ce travail est d'améliorer l'accessibilité aux workflows scientifiques et de faciliter leur création et leur réutilisation. Pour ce faire, nous proposons d'élever le niveau d'abstraction, de mettre en valeur l'expérience scientifique plutôt que les aspects techniques, de séparer les considérations fonctionnelles et non-fonctionnelles et de tirer profit des connaissances et du savoir-faire du domaine.

Les principales contributions de ce travail sont : (i) un modèle de workflows scientifiques à structure flexible, sémantique et multi-niveaux appelé "Conceptual Workflow Model", qui permet aux utilisateurs de construire des simulations indépendamment de leur implémentation afin de se concentrer sur les objectifs et les méthodes scientifiques; et (ii) un processus de transformation assisté par ordinateur pour aider les utilisateurs à convertir leurs modèles de simulation de haut niveau en workflows qui peuvent être délégués à des systèmes externes pour l'exécution.