

A Portable Approach for Bidirectional Integration between a Logic and a Statically-Typed Object-Oriented Programming Language

A Portable Approach for
Bidirectional Integration between
a Logic and a Statically-Typed
Object-Oriented Programming
Language



Pôle d'Ingénierie Informatique
ICTEAM Institute
École Polytechnique de Louvain
Université catholique de Louvain

Dissertation

A Portable Approach for Bidirectional Integration between a Logic and a Statically-Typed Object-Oriented Programming Language

Sergio Castro Mejía

8th September 2014

*Thesis submitted in partial fulfillment of the requirements
for the degree of Doctor in Engineering Sciences*

Thesis Committee:

Prof. Kim MENS (Promoter)
Prof. Charles PECHEUR (President)
Prof. Anthony CLEVE (Secretary)
Prof. Paul TARAU
Dr. Jan WIELEMAKER

INGI/UCL, Belgium
INGI/UCL, Belgium
UNamur, Belgium
Texas University, United States
Vrije Universiteit Amsterdam,
The Netherlands

A Portable Approach for Bidirectional Integration between a Logic and a Statically-Typed Object-Oriented Programming Language

© 2014 Sergio Castro Mejía
Pôle d'Ingénierie Informatique
ICTEAM Institute
École Polytechnique de Louvain
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work has been partially supported by the MoVES project of the Interuniversity Attraction Poles Programme of the Belgian Science Policy, Belgian State (2007-2008) and by a teaching assistant position at the Université catholique de Louvain, Belgium (2008-2014).

To Dharma, Diana and Parsival

Abstract

This dissertation seeks to improve on the state of the art for creating systems integrating modules written in both a logic and a statically-typed object-oriented language. Logic languages are well suited for declaratively solving computational problems that require knowledge representation and reasoning. Modern object-oriented programming languages benefit from mature software ecosystems featuring rich libraries and developer tools. The existence of several integration approaches testifies the interest of both communities in techniques for facilitating the creation of hybrid systems. In this way, systems developed in an object-oriented language can integrate modules written in a logic language that are more convenient for solving declarative problems. On the logic side, non-trivial declarative applications can take advantage of the existence of large software ecosystems such as those surrounding contemporary object-oriented languages. The combination of both paradigms allows a programmer to use the best language available for a given task.

Existing integration approaches provide different levels of abstractions for dealing with the integration concern (*i.e.*, the required interoperability in order for logic routines to access the object-oriented world, and vice versa). Some of them still require significant amounts of boilerplate code which hinders their adoption and steepens their learning curve. Others provide a high degree of integration transparency and automation which simplifies their usage. However, many of those approaches often impose strong assumptions about the architecture of a system (*e.g.*, a logic program must run embedded in an object-oriented one) thus suffering from portability issues. Furthermore, most approaches provide limited support for custom context-dependent reification of objects in the logic world and custom mappings of arbitrary logic terms to objects in the object-oriented world.

To address these problems, we introduce our portable and customisable approach for bidirectional integration between a logic and a statically-typed object-oriented language. This approach enables a transparent and (semi-) automatic communication between routines in these two worlds. In addition, it provides a customisable context-dependent mechanism for defining how artefacts in one language should be reified in the other language. A concrete implementation is provided as a portable JAVA-PROLOG interoperability framework.

To ensure portability, our framework has been made compatible with three open source PROLOG engines (SWI, YAP and XSB) by means of drivers.

We validated our approach through case studies requiring a seamless integration of declarative programs in PROLOG with object-oriented programs in JAVA.

Acknowledgements

This PhD was for me one of the major intellectual quests of my life. In retrospective, I realise how fortunate I have been in this trip for always counting with the wise guidance of a master, even in the most difficult moments of my academic adventure. Three persons played that role at different stages and in different ways. They are Kim Mens, Johan Brichau and Paulo Moura.

Johan initiated me into a world of ideas that were completely alien to me. “Logic Metaprogramming”, “Inter-Language Reflection”, “Open Unification”, all these concepts sounded to me as obscure complex words that I could never master. He had the generosity to introduce me to that new world with the unlimited patience of a kind guru towards a non-initiated. After Johan left academia seeking for adventures in the industry world with his newly created startup — where he is doing incredibly well — I had a sad feeling of losing one significant light in my research path.

I was lucky enough to find Paulo after a small while. Not only did Paulo teach me innumerable things about logic programming, he has been a main co-author during the rest of my PhD and also a good friend. I admire him and feel a deep respect for him and his never-ending enthusiasm and optimism for any enterprise he starts.

My third master was Kim. More than an advisor, Kim has been my “academic father” all these years. I recognise in him a person with a deep feeling of justice and with an honest love for science. I cannot express in words how lucky I feel for having had the privilege of being guided by him.

A PhD is just the culmination of a very long quest started many years before. I own this work to all the teachers I have had, from primary school to this moment. It would not be possible to thank all of them here, but immediately come to my mind my advisor during my master thesis professor Johan Fabry, professor Theo D’Hondt for allowing me to follow my master internship in the *PROG* lab at the *Vrije Universiteit Brussel*, professor Jacques Noyé for guiding me during my master training at the *Ecole des Mines de Nantes*, and all the inspiring teachers I had in the *Escuela Superior Politécnica del Litoral*, my original alma mater.

Special thanks to all my colleagues in the *RELEASeD* lab: Alfredo, Angela, Diego, Johan, Nicolas and Sebastian. I really enjoyed a lot both working with them and our occasional (and sometimes not so occasional) beer together.

I also would like to express my gratitude to the colleagues and friends who were so kind to proofread several chapters of this document: Bernard Lambeau and Edison Mera. Any remaining errors or omissions in this document are, of course, my sole responsibility.

This section could not be finished without a very special mention to the people who have enriched my personal life during the years that this adventure lasted. Thanks to my dear Irina, Dharma and all my family in Ecuador and Europe for all the love you gave me during these intense years.

Thanks to my friends Silvia, Geof, Richa, Cheo, Sonia, Frank, Verito, Fede, Leo, Ita and so many others for always being unconditionally there. I also own this work to you all.

My deep final thanks to Eliseo Subiela, Mario Benedetti, Peter Høeg and Pascal Fauliot for inspiring me with their creations and constantly reminding me that even all the beauty and sacrifice encountered during the intellectual quest that a PhD meant to me are, in the end, just a temporal illusion in my very short human life.

Sergio Castro Mejía
8th September 2014

The Long Path to a PhD

Qi Shang was aspiring to learn the art of archery which is known as an excellent way to reach the TAO. So he went to the venerable master *Fei Wei*. The latter told him, 'When you learn not to blink, I will teach you my art.' *Qi Shang* came back home, put himself under his wife's loom and started fixing with his eyes the movement of the shuttle without blinking. After two years of practice he was able not to blink at all, even when the point of the shuttle was touching his eye! So he went back to *Fei Wei*.

'Good,' said the Master. 'Now you need to learn to see. Catch a flea, bind it with a silk thread, and when you are able to count its heartbeat, come back to see me.' It took *Qi Shang* ten days to catch a flea, six months to manage to bind it. In one year he was able to see it as big as a plate, and in three years as big as a cart wheel. So off he ran with triumph to his master's house.

'Good,' said the Master. 'Now you need to practice your marksmanship. Hang the flea on a tree branch, retrocede fifty steps, and when you manage to transfix the flea without touching the silk thread, come back to see me.' And he gave him a bow and a quiver. It took *Qi Shang* three months to learn to bend the bow without trembling, a year to hit the trunk of the tree, two years to hit the silk thread, and three years more to be able to transfix the insect without touching the thread.

'Good,' said the old *Fei Wei*. 'You have almost finished. Now you only need to learn to do it during a storm. After that I will have nothing left to teach you.' In three years *Qi Shang* succeeded in this last feat. So he said to himself that now he had only one thing left: to try his strength against his master, to find out if he could surpass him, if he could finally occupy his place. He took his bow and arrows and went to *Fei Wei*.

The old archer, as if he was waiting for him, had gone out of the house to meet him with his bow and his sleeves rolled up. They took their positions at the opposite sides of a meadow, greeted each other silently, put the arrows on the bows and carefully took aim at each other. The bow-strings vibrated in unison, the arrows clashed and fell on the grass. Six times the arrows swished and clashed. *Fei Wei* had emptied his quiver but *Qi Shang* still had one arrow left. Ready to do anything to annihilate his rival, to finish with his master, he shot the arrow. The swishing sound of the arrow was met with a laugh, and with the little finger of his right hand the old man deflected the fatal shot, and the arrow fell down on the grass. *Fei Wei* took three steps, picked up the arrow, put it in his bow and aimed at his disciple.

Qi Shang stood still, but the arrow only touched his sash as if the master missed the aim... or spared his life. But when he made a step, his trousers fell

down. The majestic hit of the old *Fei Wei* had cut the sash.

Down fell *Qi Shang* on his knees and exclaimed, ‘Oh, great Master!’ *Fei Wei* bent down in his turn and said, ‘Oh, great Disciple!’

—**Pascal Fauliot.** Tales of wise taoists. The art of archery.

Free translation from the spanish edition.

Dedicated to Kim Mens, Johan Brichau and Paulo Moura.

Contents

1	Introduction	1
1.1	Research Context	3
1.1.1	Multi-Paradigm Programming	3
1.1.2	Language Interoperability	4
1.1.3	Logic Programming	5
1.1.4	Object-Oriented Programming	5
1.2	Problem Statement	7
1.3	Research Questions	7
1.4	Contributions	8
1.5	Supporting Publications	9
1.6	Supporting Research Artefacts	10
1.7	Roadmap	11
2	Related Work	13
2.1	Classification of our Related Work	13
2.2	Non-Embedded Logic Engines	13
2.2.1	JPL	14
2.2.2	INTERPROLOG	17
2.2.3	PDT CONNECTOR	19
2.2.4	JASPER	21
2.3	Embedded Logic Engines	23
2.3.1	TUPROLOG	23
2.3.2	P@J	26
2.3.3	JINNI	28
2.3.4	SOUL	29
2.4	Object-Oriented Libraries	33
2.5	Other Integration Approaches	35
2.5.1	Translators	36
2.5.2	Code Generation, Querying and Transformation Libraries	36
2.6	Multi-Paradigm Languages	37
2.7	Synthesis of the State of the Art	41
2.7.1	Integration Degree	41
2.7.2	State Sharing	43
2.7.3	Behavioural Integration	44
2.7.4	Multiplicity of Environments	46
2.7.5	Customisability	46
2.7.6	Portability	47
2.8	Comparison of Integration Features	48

2.8.1	Conventions	48
2.8.2	Interpretation	48
2.8.3	Relevance of Features	48
2.8.4	Dependency Correlation	50
2.8.5	Other Correlations	50
2.9	Chapter Summary	50
3	Case Studies	53
3.1	The London Underground	53
3.1.1	PROLOG-Side Artefacts	53
3.1.2	JAVA-Side Artefacts	58
3.1.3	Discussion	60
3.2	An Application for Querying Geographical Data	61
3.2.1	JAVA-Side Artefacts	62
3.2.2	PROLOG-Side Artefacts	62
3.3	Chapter Summary	69
4	A Conceptual Model for Bidirectional Integration between a Logic and a Statically-Typed Object-Oriented Language	71
4.1	Design Trade-Offs	72
4.2	Defining a Common Model	72
4.3	Capturing Object State: The Critical Integration Point	73
4.3.1	Object Reification	73
4.3.2	Opacity of the Representation	74
4.3.3	Object Identity Preservation	76
4.3.4	Reference Life Span	77
4.3.5	Discussion on Object State Integration Dimensions	78
4.4	Inter-Language Conversions as a Portable Integration Technique	79
4.4.1	Specification of Inter-Language Conversion Functions	79
4.4.2	Application of Inter-Language Conversion Functions	81
4.4.3	Formalisation	82
4.4.4	Implications and Limitations	83
4.4.5	Is the Paradigm Leak Mostly Solved ?	84
4.5	Behavioural Integration	84
4.5.1	From the Logic to the Object-Oriented Language	85
4.5.2	From the Object-Oriented to the Logic Language	88
4.5.3	Error Handling	89
4.5.4	Discussion on Behavioural Integration	90
4.6	Chapter Summary	91
5	An Integration Framework Architecture	93
5.1	A High Level View	93
5.1.1	Core Components	93
5.1.2	Dynamic Interactions from the JAVA Perspective	94
5.1.3	Interoperability Drivers	95
5.1.4	Dynamic Interactions from the PROLOG Perspective	95

5.1.5	Discussion of the High-Level Architecture	95
5.2	Plumbing the Paradigm Leak with LOGTALK	95
5.3	The Concrete Architecture of our Integration Framework	97
5.3.1	LOGTALK Dependency	97
5.3.2	The JPC Library	98
5.3.3	The LOGICOBJECTS Library	98
5.3.4	JPC Drivers	99
5.4	Chapter Summary	99
6	Inter-language Conversions between Prolog and Java	101
6.1	Analysis	101
6.1.1	Defining Inter-Language Conversion Functions	102
6.1.2	Conversion-Type Inference	104
6.1.3	Factories	105
6.1.4	Categorising Conversion Artefacts	106
6.1.5	Applying Conversions	106
6.1.6	Context as Converter Discriminator	106
6.2	Design	107
6.2.1	Design Trade-Offs	107
6.2.2	An Embedded PROLOG Database	107
6.2.3	A Reification of PROLOG Terms in JAVA	108
6.2.4	The JPC Context	108
6.2.5	A Type-Based Categorisation Framework	111
6.3	Inter-Language Conversions	117
6.3.1	Primitives Conversions	117
6.3.2	Typed Conversions	118
6.3.3	Multi-Valued Conversions	119
6.3.4	Generic Types Support	119
6.3.5	Inference of Conversion Target Types	120
6.3.6	Custom Conversions	121
6.3.7	Term-Quantified Converters	122
6.4	Discussion and Limitations	122
6.5	Chapter Summary	123
7	A Portable Context-Dependent Integration Library	125
7.1	From JAVA to PROLOG	125
7.1.1	Behavioural Integration from JAVA	125
7.1.2	JAVA-Side Reference Management API	133
7.2	From PROLOG to JAVA	138
7.2.1	Behavioural Integration from PROLOG	139
7.3	Chapter Summary	145
8	A Bidirectional Linguistic Integration Framework	147
8.1	From PROLOG to JAVA with LOGICOBJECTS	147
8.1.1	Inferring the Receiver of the Method	148
8.1.2	Inferring the Name of the Method	148

8.1.3	Transforming Predicate Parameters to Method Parameters	149
8.1.4	Interpreting Unbound Logic Variables	149
8.1.5	Interpreting the Return Value of a Method	149
8.1.6	Error Handling	151
8.1.7	JAVA Expressions in PROLOG	151
8.1.8	Explicit Reflection	153
8.1.9	Choosing the Right Constructs	154
8.2	From JAVA to PROLOG with LOGICOBJECTS	156
8.2.1	Inferring the Receiver on the Logic Side	156
8.2.2	Inferring the Predicate Name	157
8.2.3	Transforming Method Parameters to Predicate Parameters	158
8.2.4	Passing Unbound Variables as Predicate Arguments . . .	158
8.2.5	Interpreting a Query Solution as an Object	159
8.2.6	Non-Determinism Support	160
8.2.7	Error Handling	161
8.2.8	Instantiating Symbiotic Classes	162
8.2.9	Auto-loading LOGTALK objects	163
8.2.10	Macros	163
8.2.11	Integration with plain PROLOG	163
8.2.12	Choosing the Right Constructs	164
8.3	Discussion	165
8.4	Chapter Summary	165
9	Validation	167
9.1	Validation of the Integration from the Object-Oriented Side . . .	167
9.2	Validation of the Integration from the Logic Side	170
9.2.1	Configuring the Default Conversion Context	170
9.2.2	The city/1 LOGTALK Object	172
9.2.3	Loading Geographical Data from an OSM File	172
9.2.4	Opening the MAPQUERY Browser	173
9.2.5	Zooming and Moving the Map	173
9.2.6	Querying and Drawing in the Map	174
9.2.7	Wrapping up	174
9.3	Portability Validation	175
9.3.1	HYDRA Components	175
9.3.2	Configuring the PROLOG Engine Creation	175
9.3.3	Managing Drivers	176
9.3.4	Managing PROLOG Sessions	177
9.3.5	Management of PROLOG Queries	178
9.3.6	Query Edition Options	179
9.3.7	Presenting Query Results	179
9.3.8	Loading Files	180
9.3.9	LOGTALK Support	180
9.4	Threats to Validity	181

9.4.1	Limits regarding the Transparency and Automation of the Integration	181
9.4.2	Limits of Type Inference	182
9.4.3	Lack of Causal Connection in the Reification of Foreign Artefacts	182
9.4.4	Complexity on Understanding and Debugging Type-Guided Conversions	183
9.4.5	Context-Depending Conversions Issues	183
9.4.6	Performance Issues	183
9.4.7	Portability Issues	183
9.5	Discussion	183
9.5.1	Transparency and Automation	184
9.5.2	Causal Connection vs. Inter-Language Conversions	184
9.5.3	Complexity of our Approach	184
9.5.4	Performance Optimisation Paths	185
9.6	Integration Features Revisited	185
9.7	Cross-Fertilisation from Several Domains	187
9.8	Chapter Summary	187
10	Conclusions	189
10.1	Our Research Questions Revisited	189
10.2	Future Work	191
10.2.1	Inter-Language Reflection	191
10.2.2	Muti-threading support	191
10.2.3	Development of New Drivers	192
10.2.4	Improvement of the JPC Embedded PROLOG Database	192
10.2.5	Development of new Case Studies	192
10.3	Conclusions	192
	Bibliography	195
	List of Terms	210
	Acronyms	211

1 Introduction

Writing is good, thinking is better.
Cleverness is good, patience is better.
—**Hermann Hesse, Siddhartha**

This dissertation seeks to improve on the state of the art for creating systems that integrate modules written in a logic and a statically-typed object-oriented language.

Logic languages are convenient for reasoning over problems of declarative nature, such as expert and planning systems [116, 123]. However, it is often difficult to develop complete applications that require e.g. GUIs, heavy numerical computations, or low-level operating system and network access [13, 14, 138]. On the other hand, object-oriented languages have demonstrated their usefulness for modelling a wide range of concepts (*e.g.*, GUIs) found in many business scenarios [90]. The availability of continuously growing software ecosystems around modern object-oriented languages, including advanced IDEs and rich sets of libraries, has significantly contributed to their success.

Non-trivial applications can profit from implementing their components, or even distinct routines of the same entity, in the language that is most appropriate for expressing them [13, 14, 47, 89, 104, 138]. However, the integration of programs or routines written in different languages is not trivial when such languages belong to different paradigms [9, 66].

For the case of object-oriented and logic-programming languages in particular, several approaches have been developed to allow bidirectional interoperability between a logic language and an object-oriented language. Our survey of the state of the art in section 2.8 reveals that most of these approaches, however, are complex to apply. They often require a significant amount of boilerplate code that obscures the core purpose of the application components and tangles it with the integration concern.

Other approaches [10, 40, 47] have attempted to reach a seamless interoperability by providing integration support at the linguistic level. This technique, often referred as *linguistic symbiosis* [75], allows to invoke foreign routines as if they were defined in the native language [66] and represents foreign objects as any other native artefact [144].

However, many of these existing approaches impose strong assumptions regarding the architecture of the system (*e.g.*, assuming that the logic program

is running embedded in the object-oriented programming environment). This creates a significant gap between what the approach can offer and what is actually required in several real business scenarios. In addition, existing approaches provide limited support for defining a custom context-dependent reification of objects in the logic world and establishing mappings of arbitrary logic terms to objects in the object-oriented world. Furthermore, we have found that most existing work accomplishing a bidirectional symbiotic integration targets a dynamically-typed language on the object-oriented side. Other approaches targeting a statically-typed language tend to have a limited functional scope (*e.g.*, querying, generating or transforming an object-oriented program from a logic program [40, 42]) or are unidirectional.

Our research hypothesis is therefore that considerable room for improvement still exists in the integration of hybrid systems consisting of logic and statically-typed object-oriented modules. More advanced interoperability abstractions should be designed starting from the lessons learned in previous experiences in this and similar research domains. These abstractions should be made portable and compatible with different system architectures and execution environments (*e.g.*, either as an embedded logic engine or as a separate program).

The research presented in this dissertation aims to improve on current techniques for reducing, or eliminating completely in certain scenarios, the amount of integration boilerplate code that currently needs to be written. We target both the object-oriented and the logic perspective in applications that combine object-oriented and logic abstractions.

Our vision is to provide a conceptual framework and its corresponding implementation that simplifies the creation of hybrid systems composed of both logic and statically-typed object-oriented code. This simplification is accomplished by means of declarative programming techniques, where integration aspects are either declaratively specified or subject to an inference process (*e.g.*, based on reflection). This reduces and facilitates the programming effort required to build such hybrid systems and makes the final application code more readable and maintainable.

We introduce a new linguistic integration library called LOGICOBJECTS that epitomises our conceptual framework. Our library targets the JAVA [62] and PROLOG [35] languages. JAVA guarantees, to a certain degree, the portability and deployability of an application. PROLOG, on the other hand, facilitates the exploitation of decades of research in logic programming [13, 14].

Our integration library allows for a fine-level of control over how JAVA objects are reified as PROLOG terms and how PROLOG terms are expressed as JAVA objects. At the same time, it makes use of the JAVA type system to infer, when possible, the most appropriate mapping for inter-language artefacts. We reach full portability by not coupling our implementation to a particular PROLOG engine. Instead, by following an *abstraction approach* as described by Wielemaker et al. [140], it interacts with an abstract representation of a PROLOG virtual machine. This virtual machine abstraction can easily operate on any concrete underlying PROLOG engine by means of drivers. In this

way, we empower programmers working on hybrid PROLOG-JAVA systems to choose the most appropriate PROLOG engine according to the requirements of their problem (*e.g.*, availability of libraries, performance, or operating system compatibility). At the same time, we achieve reusability and portability among PROLOG dialects, even for the foreign JAVA components with which the PROLOG components interact.

Our implementation of this integration library is validated by means of a few case studies illustrating both sides of the symbiotic integration.

In the next section we provide a more detailed discussion on our research context in order to position our work against similar research domains. We then continue in the problem statement section by detailing the issues to be tackled in this research. Afterwards we formulate the main research questions that will guide our development throughout this dissertation and present our main contributions in section 1.4. We conclude this introductory chapter with a roadmap which serves as a guide to the reader through the rest of this dissertation.

1.1 Research Context

In this section we describe the distinct domains that form our research context.

1.1.1 Multi-Paradigm Programming

Different programming paradigms introduce distinct programming concepts and abstractions. In certain scenarios, combining those concepts is highly desirable [70]. The most common approaches to achieve this are either by means of a multi-paradigm programming language encompassing all the desirable concepts, or by using an integration technique allowing to interoperate modules written in different languages.

Both techniques present significant limitations. Integrating distinct paradigms in a unique programming language is far from trivial, given the complex interactions between their features [70]. In the same way, integrating modules written in languages belonging to different paradigms is hindered by the inherent complexity of mapping concepts between them (*e.g.*, there may be concepts that exist in one language but not in the other one). This integration problem is particularly present between a logic and an object-oriented language [14, 45, 46, 73].

In this dissertation we follow the approach of building a multi-paradigm system by means of the integration of modules written in distinct programming languages. This technique has the benefit that, when applied to existing languages, a programmer may be able to take advantage of the software ecosystems surrounding them.

1.1.2 Language Interoperability

Language interoperability as a strategy for multi-paradigm programming can take multiple forms. In this section we list some of the most common ones found in literature.

Inter-Process Communication

In the past, many approaches have attempted to reach a seamless interoperability between heterogeneous components by means of a common communication protocol [37, 63, 64, 120]. This protocol can be either binary (*e.g.*, CORBA [64]) or text based (*e.g.*, XML [85] or JSON [39] web services [2]). However, these techniques usually require a developer to take care of a considerable amount of details concerning the integration in any non-trivial application. This is partly caused by the significant amount of boilerplate code that integrators are forced to write and maintain [72].

Language Bindings

Other approaches are based on the automatic generation of the glue code required to accomplish the integration (*e.g.*, SWIG [50]). In such approaches, the generated code wraps a component so that it can be used from the other programming language. However, generating this glue code often demands its own boilerplate code (*e.g.*, explicit specification of the routines to interface) and introduces the complexity of keeping the generated glue code synchronised with changes in the code to integrate.

Linguistic Integration

Some approaches have attempted to solve the integration problem by defining mechanisms for programs to transparently invoke routines defined in the foreign language as if they were defined in their own language [9, 47, 66]. In addition to this behavioural integration, objects from one language should be understood in the other [144].

A representative example of this tight integration is the .NET framework [134]. Programs compiled for the .NET common language runtime can reach a seamless integration with other languages compiled for the platform. In this way, programs written in different languages are converted into a common binary representation which allows them to communicate with each other. For example, a class in one language can inherit from a class in another language or call a method in the original class. Also, an instance of a class can be passed to a method of a class written in a different language.

From the linguistic perspective, this high-level integration is relatively easy to accomplish since the participating languages belong to the same paradigm: they are all imperative and object-oriented.

However, integrating languages belonging to different paradigms (*e.g.*, a statically-typed imperative object-oriented language with a dynamically-typed

logic language) presents more complex challenges. This occurs when attempting to map artefacts or concepts that exist in one paradigm but do not exist in the other one. This problem has been referred to as *paradigm leaking* [9, 66], indicating the leaking of concepts when jumping from one programming paradigm to another.

Putting all previous concepts together, this dissertation focuses on *linguistic integration* as a portable *language interoperability* technique for reaching *multi-paradigm programming* between a dynamically-typed logic and a statically-typed object-oriented language.

1.1.3 Logic Programming

Logic Programming (LP) is a programming paradigm making use of logic directly as a programming language. Starling and Shapiro [123] define *logic* as “the foundation for deducing consequences from premises; for studying the truth or falsehood of statements given the truth or falsehood of other statements; for establishing the consistency of one’s claims; and for verifying the validity of one’s arguments”. In logic programming, a *program* is conceived as a logical theory (*i.e.*, a set of axioms). Any *computation* is a constructive proof of a goal statement from the program.

In other words, a logic program is a declarative specification of **what** the problem to be solved is. This in contrast to the traditional view of imperative languages where a program is defined as a procedural specification of **how** the problem is to be solved [55].

Therefore, by means of logic programming it is often possible to write in a very concise and clear style programs that may be lengthy and cumbersome in other programming languages. This is particularly the case in knowledge-intensive programs.

The concrete logic language we selected to instantiate our approach is PROLOG. From the existing logic programming languages, it is the oldest and perhaps the most well-known language. A particular advantage of PROLOG over more contemporary logic languages (*e.g.*, MINIKANREN [12]) is that there is already a considerable body of PROLOG libraries for logic reasoning. These existing PROLOG libraries are the result of decades of research in logic programming, are well tested, and some of them have been successfully employed in industrial settings for years (*e.g.*, the CHR library [121]). In addition, there exist many high-quality open-source implementations, some of which are very fast.

A disadvantage of PROLOG, however, is its comparatively poor software ecosystem regarding general-purpose libraries, such as the ones available in other languages like JAVA or the .NET platform.

1.1.4 Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm where the decomposition of a system is based upon the notion of encapsulated objects

that are able to respond to particular messages. One of its core goals is to manage the complexity of massive software-intensive systems [5]. Modern languages following this programming paradigm are often surrounded by rich software ecosystems such as advanced programming environments and reusable components [53].

Object-oriented languages provide mechanisms for data and code encapsulation [107], and code reuse mechanisms such as class or prototype based inheritance [6] or composition [71]. Meyer observes that the term “object-oriented” is not a boolean condition and that different programming languages may have different degrees of “object-orientation” [90]. Furthermore, the concept of “object-oriented programming” is orthogonal to “logic programming” [104], since there exist languages that integrate both paradigms (*e.g.*, LOGTALK [95]).

As in other programming paradigms, object-oriented languages can either have a dynamic or a static type system. A *type system* is “a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute” [111]. While in dynamically-typed languages the type of constructs is checked at runtime [108], in statically-typed languages the type of every expression can be determined by static program analysis [15]. Both systems have their advantages and shortcomings [88], and it is not our intention here to claim superiority of one over the other. However, in the context of our research, the additional type data provided by statically-typed languages makes them more amenable to integration techniques based on static analysis (*e.g.*, the inference of the best integration strategy may be guided by declared types in the object-oriented language). Also, some of the most widely-used object-oriented languages (like JAVA) are statically typed.

Therefore, the scope of this dissertation concerns the integration of a logic language with a statically-typed object-oriented language. In this scope, we refer to an object-oriented language as a language adhering to object-oriented principles and not integrating logic programming concepts.

The concrete object-oriented language we chose to instantiate our approach is JAVA. There are multiple reasons for that choice. Some of them are:

- The significant size and quality of the JAVA software ecosystem (*e.g.*, software libraries, development tools and even a considerable number of emerging languages running on the JAVA virtual machine).
- The high portability of JAVA across different platforms, including mobile devices.
- The existence of several approaches targeting JAVA attempting to integrate it with a logic language, since we can learn from and build upon those previous experiences.

1.2 Problem Statement

Even though previous approaches have made significant steps towards providing language support for integrating a logic and an object-oriented language, there are still many open questions and room for improvement. We argue that (1) existing techniques should be generalised and made portable to common architectures. Furthermore, (2) new techniques, particularly those profiting from the additional type data provided by statically-typed systems, should be developed to simplify the problem of mapping concepts between the two language paradigms. Finally, (3) a programmer should not be constrained to a static and fixed integration strategy. Instead, integration strategies should be scoped to a particular context and a programmer should be able to easily switch between distinct strategies, even in the same application if required.

Our position is that having a dynamic, context-dependent integration policy that infers the best integration strategy based on static type analysis and programming conventions will significantly reduce the amount of boilerplate integration code that currently needs to be written. In this way, the implementation and maintenance of hybrid logic and object-oriented systems will be considerably facilitated.

1.3 Research Questions

To solve the research problem mentioned above, we need to provide an answer to the following research questions:

RQ.1 What is the degree of integration provided by existing interoperability approaches and what is the architecture of such approaches?

The design of an integration solution should start by an analysis of the advantages and shortcomings of existing interoperability tools and techniques. This study should allow to determine which is the architecture that existing tools target and which are the assumptions under which they are developed.

RQ.2 What are the linguistic integration features required for hybrid logic and object-oriented systems?

Understanding the required integration features is a core aspect of our problem since it will motivate the rest of our research objectives.

RQ.3 From the conceptual point of view, what are the high-level steps in order to accomplish a linguistic integration?

Once the desirable linguistic integration aspects have been defined, a high-level analysis is required to determine which are the implementation-independent steps to accomplish those goals.

RQ.4 To what extent is it possible to profit from the additional type data provided by a statically-typed language to accomplish an automatic and transparent integration?

An integration approach targeting a statically-typed object-oriented language should profit, if possible, from the additional data gathered by means of static type analysis. This data could guide the inference of the best integration technique to accomplish.

RQ.5 How can an integration approach be customised to a particular user-defined context?

A programmer should not be constrained to a static integration strategy. Instead, the integration strategy could dynamically change at runtime if needed, or multiple integration strategies could co-exist in the same application.

These research questions define the requirements that will guide the development of this dissertation.

1.4 Contributions

This section highlights the main contributions of this dissertation.

A Conceptual Approach for Achieving a Bidirectional Integration between a Logic Language and a Statically-Typed Object-Oriented Language

We provide a conceptual model describing the integration of a logic language and a statically-typed object-oriented language. This conceptual model is general enough to be applied to other languages than the ones we chose to validate our work (*i.e.*, PROLOG and JAVA).

An abstract integration framework between Prolog and Java

We design the architecture of an abstract framework for the integration of a concrete logic language (PROLOG) with a concrete statically-typed object-oriented language (JAVA).

A Portable Context-Dependent Integration Library for Prolog and Java

Part of our validation is a hybrid library accomplishing a context-dependent bidirectional integration between PROLOG and JAVA. The purpose of this library is to serve as a portable, general-purpose integration layer between those languages. This library attempts to achieve a full portability by decoupling an abstract representation of a PROLOG virtual machine and a concrete implementation of a PROLOG engine. Although this library provides basic interoperability support, more advanced integration frameworks could be implemented on top of this layer.

A Bidirectional Linguistic Integration Framework for Prolog and Java

A concrete implementation of a more high-level integration framework which builds upon the lower level integration library. This framework provides bidirectional transparent and (semi-)automatic integration between PROLOG and JAVA. As discussed in our related work, logic engines embedded in object-oriented environments are typically the ones providing more advanced integration features from the linguistic perspective. For portability reasons, one of our main goals has been to reach this level of linguistic integration in more complex and common scenarios where, for example, logic engines run as a separate program.

A Validation of our Integration Framework

We validate our implementation by means of several case studies focusing on different aspects of the integration.

1.5 Supporting Publications

The following authored publications support the key ideas in this dissertation:

- **JPC: A Library for Categorising and Applying Inter-Language Conversions Between Java and Prolog** [29].

Sergio Castro, Kim Mens and Paulo Moura.

Submitted to Science of Computer Programming: Experimental Software and Toolkits (EST 6).

This paper describes the core of our techniques for encapsulating, categorising and applying JAVA-PROLOG conversion routines. It introduces a new framework, the JPC library, serving as a development tool for both programmers willing to categorise context-dependent conversion constructs in their JAVA-PROLOG systems, and for architects implementing frameworks providing higher-level abstractions for better interoperability between these two languages.

- **Customisable Handling of Java References in Prolog Programs** [28].

Sergio Castro, Kim Mens and Paulo Moura.

To appear in Proceedings of the International Conference on Logic Programming (ICLP 2014).

This paper discusses the problem of the proper representation of references to JAVA objects on the PROLOG-side. Multiple dimensions are considered, including reference representation, opacity of the representation, identity preservation, reference life span, and scope of the inter-language conversion policies. We generalise and build upon existing representation patterns of foreign references in PROLOG, and take inspiration from similar inter-language representation techniques found in other domains.

- **LogicObjects: A Portable and Extensible Approach for Linguistic Symbiosis between an Object-Oriented and a Logic Programming Language** [18].

Sergio Castro.

Proceedings of the International Conference on Logic Programming, Doctoral Consortium (ICLP DC 2013).

This paper presents a high-level overview of our approach to bidirectional integration of an object-oriented and a logic language. It overviews our initial results, describes related work and sets the general research roadmap.

- **LogicObjects: Enabling Logic Programming in Java through Linguistic Symbiosis** [27].

Sergio Castro, Kim Mens and Paulo Moura.

Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL 2013).

This paper refines our previous work presenting our full solution for linguistic integration from the object-oriented perspective (*i.e.*, JAVA). We introduce improved mechanisms for automatic adaptation of logic routine results in the object-oriented world, a context-dependent mapping of JAVA objects to multiple representations in PROLOG, and a general mechanism for expressing JAVA objects in a convenient logic representation even in pure object-oriented programs.

- **LogicObjects : A Linguistic Symbiosis Approach to Bring the Declarative Power of Prolog to Java** [26].

Sergio Castro, Kim Mens and Paulo Moura.

Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAM-SE 2012).

This paper presents an initial prototype for linguistic integration between an object-oriented and a logic language. This work focuses only on the object-oriented perspective and proposes an approach based on runtime code generation and byte code instrumentation.

1.6 Supporting Research Artefacts

In addition to the 5 publications mentioned in the previous section, a set of software artefacts was developed in the context of this work. As they evolved over time, they became independent open-source projects, each of them implementing and validating a different aspect of our research. These artefacts are listed below:

- **Java-Prolog Connectivity (JPC)** [22]

JPC is a JAVA-PROLOG interoperability library providing different levels of integration abstractions. One of the core JPC components consists of a module in charge of inter-language conversions between JAVA and

PROLOG artefacts. In many aspects, the implementation of this module is inspired by Google’s GSON [61], a state of the art library for accomplishing conversions between JAVA and JSON artefacts. JPC is currently compatible with three of the most popular open-source PROLOG engines: SWI [142], YAP [38] and XSB [126] PROLOG.

- **LogicObjects** [23]

LOGICOBJECTS extends JPC by providing the necessary means for integrating JAVA and PROLOG programs at the linguistic level. In this way, routines between the two worlds can be invoked transparently and foreign language artefacts can be referenced as local ones.

- **JConverter** [20]

JCONVERTER generalises the inter-language conversion techniques originally defined in JPC. It defines an architectural pattern for accomplishing type-guided conversions between arbitrary artefacts in JAVA.

- **JGum** [21]

JGUM allows to create and manipulate different kinds of categorisations in JAVA. The notion of categories plays a major role in computer programming. However, for statically-typed languages it is often not trivial (or not possible at all) to modify at runtime the properties of existing categories, or create new categories on the fly. With JGUM, a programmer can define and manipulate ad hoc categorisations, as well as easily organise categories relying on existing JAVA categorisation mechanisms, such as classes and packages. JGUM categorisations are at the core of other libraries implemented in the context of this dissertation.

1.7 Roadmap

The main contribution of this dissertation is *a conceptual model of a bidirectional integration framework between a logic language and a statically-typed object-oriented language, instantiated by a concrete implementation of a portable bidirectional integration framework for PROLOG and JAVA.*

Below we summarise the chapters of this dissertation.

Chapter 2: Related Work provides an overview of related work. Although our main interest is in language interoperability techniques based on linguistic integration between logic and object-oriented languages, we also survey other works in similar domains that provide significant inspiration. From this survey we define which are the desirable features in a linguistic integration problem between our languages of interest.

Chapter 3: Case Studies provides concrete examples of problems that can be naturally solved with a hybrid application. These examples allow us to illustrate the complexity of the integration both from the object-oriented and logic programming perspective. The problems discussed here will be referenced in the chapters that follow.

Chapter 4: A Conceptual Model for Bidirectional Integration between a Logic and a Statically-Typed Object-Oriented Language provides an implementation-independent conceptual model for the integration of logic and statically-typed object-oriented languages.

Chapter 5: An Integration Framework Architecture provides a description of the architecture of an integration framework following our conceptual model.

Chapter 6: Inter-language Conversions between Prolog and Java describes a library accomplishing inter-language artefact conversions between PROLOG and JAVA.

Chapter 7: A Portable Context-Dependent Integration Library describes a library accomplishing a portable context-dependent bidirectional integration between PROLOG and JAVA. This library provides the basis for a more advanced integration framework introduced next.

Chapter 8: A Bidirectional Linguistic Integration Framework discusses a concrete implementation of our more high-level linguistic integration framework for PROLOG and JAVA.

Chapter 9: Validation provides a validation of our approach by means of describing how solutions to the problems introduced in chapter 3 can be easily implemented with the constructs we provide. We illustrate how these solutions are more transparent and automatic in comparison to other techniques. We finalise this chapter with a discussion detailing the strengths and weaknesses of our approach.

Chapter 10: Conclusions wraps up the dissertation by restating the contributions in a fine-grained and more technical manner. The chapter discusses the current limitations of our work and suggests future avenues in which research in integration frameworks could be further developed.

2 Related Work

Ay, deeper thoughts than these, though undefined,
start in the reflective soul at sight of thee.
—**Arthur Wentworth Eaton, The lotus of the Nile**

Several techniques have been proposed in the past to integrate logic and object-oriented programs. In this section we present an overview of the state of the art. We include in our survey not only existing techniques targeting statically-typed object-oriented languages as in our approach, but also dynamically-typed object-oriented languages, since they were a significant source of inspiration.

In section 2.7 we summarise the most significant properties we found in the different surveyed approaches. Where relevant, we highlight the features that will help us to identify later which are the relevant integration dimensions in the context of our research problem.

2.1 Classification of our Related Work

There are many possible ways to classify our related work. In general, this section is structured according to the execution environment of the logic and object-oriented language. More specifically, whether the logic program runs embedded in the object-oriented program or as a separate program.

Given that available integration techniques usually depend on a concrete architecture, this division allows to facilitate the comparison of similar approaches. However, other interesting techniques not classifiable under one of these criteria (*e.g.*, object-oriented libraries for logic languages such as XPCE) are also discussed along this chapter.

2.2 Non-Embedded Logic Engines

Integrating an object-oriented program with a non-embedded logic engine presents particular challenges. For example, object references from the object-oriented world should be symbolically represented on the logic side and the garbage-collector processes should be somehow coordinated.

As an advantage, native logic engines are usually significantly more performant than engines embedded within object-oriented environments. In addition, they tend to meet more the official and de facto standards, which empowers them to reuse a considerable amount of well-proven logic libraries. Significant well-known open-source examples are SWI, YAP, and XSB PROLOG among others.

2.2.1 JPL

JPL [119] is a JNI-based [82] JAVA-PROLOG interoperability library compatible with SWI [142] and YAP [38] PROLOG.

From Java to Prolog with JPL

From the JAVA perspective JPL provides both low and high level interfaces. Since our goal is to review advanced integration features, we will focus our discussion on the high-level interface only. JPL allows JAVA programmers to execute PROLOG queries either from their textual representation (*e.g.*, code snippet 2.1) or using a structured reification of PROLOG terms (*e.g.*, code snippet 2.2).

```
1 Query q = new Query( "teacher_of(aristotle, alexander)" );
```

Snippet 2.1: A textual query using JPL.

```
1 Term goal = new Compound( "teacher_of", new Term[]{new Atom("aristotle"),↵
    new Atom("alexander")} );
2 Query q = new Query( goal );
```

Snippet 2.2: A structured query using JPL.

A query object implements the `java.util.Enumeration` interface. By means of those implemented methods, all solutions to a query can be traversed. The JPL user guide defines a *query solution* as “a collection of *bindings*, each of which relates one of the Variables within the Query’s goal to a Term representation of the PROLOG term to which the corresponding PROLOG variable was bound by the proof”.

At the implementation level, a solution is represented by means of an instance of `java.util.Hashtable`, which represents bindings of variables (variable names) to terms. An example of this is illustrated in code snippet 2.3. A new query, with an unbound logic variable, is declared on line 2. The query is traversed using the methods implemented from the `Enumeration` interface (lines 3–7). For each solution, the term bound to the logic variable is obtained (line 5) and printed in the console (line 6).

```
1 Variable X = new Variable();
2 Query q = new Query( "teaches", new Term[]{new Atom("aristotle"),X});
3 while ( q.hasMoreElements() ) {
4     Hashtable binding = (Hashtable) q.nextElement();
5     Term t = (Term) binding.get(X.name());
```

```

6      System.out.println(t);
7  }

```

Snippet 2.3: Traversing all the solutions of a query using JPL.

In addition to methods to traverse all solutions to a query, there are auxiliary methods to obtain only one solution (useful for deterministic queries) or to obtain all the solutions in one single method call.

From Prolog to Java with JPL

JPL also provides an interface for interacting with JAVA from within PROLOG. This interface is completely dynamic and allows to interact from PROLOG with any class that can be found on the JAVA classpath.

The four main predicates for interacting with JAVA are `jsp_new/3` (instantiating a JAVA class), `jsp_call/3` (invoking a JAVA method), `jsp_set/3` (setting a field of an object) and `jsp_get/3` (reading a field of an object).

Object references returned by JAVA methods are treated as opaque handles on the PROLOG-side. Two reference handles are considered equal (*i.e.*, unify) if-and-only-if they refer to the same object within the JVM. These handles contain a distinctive atom in order to exploit SWI-PROLOG's atom garbage collection. In this way, "when an object reference is garbage-collected in PROLOG, the JVM garbage collector is informed, so there is sound and complete overall garbage collection of JAVA objects within the combined PROLOG+JAVA system".

The example shown in code snippet 2.4, extracted from the JPL documentation, illustrates a typical usage of the JAVA-PROLOG predicates. The objective is to show in a table, using the Swing library [51], the current PROLOG flags and their values.

The `findall/3` meta-predicate (lines 1–8) queries a PROLOG list of an array of JAVA strings. Each array has two elements, the first being the name of a PROLOG flag and the second its value. The PROLOG list of JAVA arrays is then converted to a bidimensional JAVA array and its value bound to the `Ac` variable (line 9). On line 10, a JAVA array with the headers of the table is created from a PROLOG list of atoms by means of the auxiliary predicate `jpl_datums_to_array/2`. This array is bound to the PROLOG variable `Ah`. A new Swing `JFrame` component is instantiated and bound to the variable `F` on line 11. Its `contentPane` property is accessed on line 12 and bound to the variable `CP`. A `JTable` component is instantiated from the JAVA arrays containing the content of the table (`Ac` PROLOG variable) and its header (`Ah` PROLOG variable). The calls below put the table in a scroll pane (line 14) and the scroll pane in the content pane of the `JFrame` component (line 15). Finally, the `JFrame` component is resized (line 16). The table is shown in figure 2.1¹.

```

1 findall(
2     Ar,
3     (    current_prolog_flag( N, V),

```

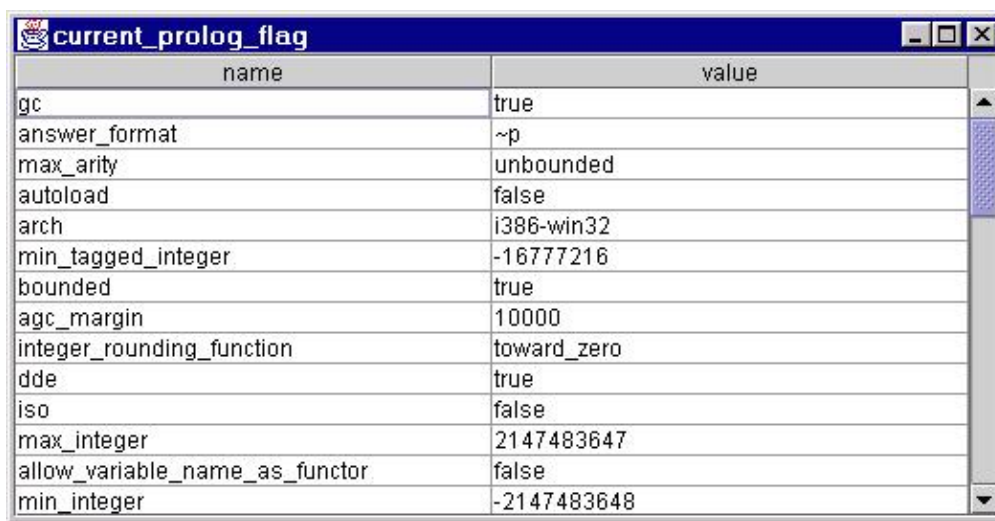
¹Image taken from the JPL user manual at http://www.swi-prolog.org/packages/jpl/prolog_api/overview.html

```

4      term_to_atom( V, Va),
5      jpl_new( '[Ljava.lang.String;', [N,Va], Ar)
6  ),
7  Ars
8  ),
9  jpl_new( '[[Ljava.lang.String;', Ars, Ac),
10 jpl_datums_to_array( [name,value], Ah),
11 jpl_new( 'javax.swing.JFrame', ['current_prolog_flag'], F),
12 jpl_call( F, getContentPane, [], CP),
13 jpl_new( 'javax.swing.JTable', [Ac,Ah], T),
14 jpl_new( 'javax.swing.JScrollPane', [T], SP),
15 jpl_call( CP, add, [SP,'Center'], _),
16 jpl_call( F, setSize, [600,400], _)

```

Snippet 2.4: From PROLOG to JAVA example using JPL.



name	value
gc	true
answer_format	~p
max_arity	unbounded
autoload	false
arch	i386-win32
min_tagged_integer	-16777216
bounded	true
agc_margin	10000
integer_rounding_function	toward_zero
dde	true
iso	false
max_integer	2147483647
allow_variable_name_as_functor	false
min_integer	-2147483648

Figure 2.1: Showing a Swing table with PROLOG properties by means of JPL.

Note that in the previous example the programmer does not have to worry about the life cycle of JAVA references on the PROLOG-side (*e.g.*, all references to JAVA arrays). Instead, JAVA references remain valid as long as the PROLOG atom contained in the reference handler on the PROLOG-side is not garbage-collected (*i.e.*, the atom is referenced somewhere in the PROLOG database).

JPL Summary

JPL, although still providing distinct levels of integration abstractions (*e.g.*, its low-level vs. its high-level interface) still demands a significant explicit usage of constructs regarding the integration concern. The PROLOG programmer cannot avoid to explicitly refer to predicates abstracting JAVA concepts such as creating a new instance or accessing an object property. In the same way, a JAVA program integrated with PROLOG has tangled code related to the integration concerns (*e.g.*, manipulating a PROLOG engine, or explicitly creating a query or resolving logic variable bindings).

The JPL mechanisms for making the two garbage collectors collaborate was not found in any other PROLOG engine non-embedded in JAVA which we studied. However, although powerful, it is the only mechanism available, forcing the

programmer to define ad-hoc alternatives if a different management of foreign references is required (*e.g.*, keeping a PROLOG handler to a JAVA reference valid as long as it is not explicitly invalidated).

Although quite efficient in comparison to other approaches, JPL programs are rather difficult to debug, which is a recurrent reported problem on the SWI [125] and YAP [146] users mailing lists. This is not a problem related to the underlying PROLOG engines, but to the technique employed for the JAVA–PROLOG communication (JNI).

Again in comparison to other approaches, another disadvantage of JPL is the limited control a JAVA programmer has over a PROLOG engine. For example, there is currently no way to stop a PROLOG engine, to restart it, or to abort an executing query.

2.2.2 InterProlog

INTERPROLOG [13, 14] allows JAVA programmers to pass objects to PROLOG either by reference (as in JPL) or as serialised streams of bytes. At the time of writing, the publicly available version is socket-based and compatible with XSB only. Previous versions have also been compatible with SWI and YAP PROLOG.

From Prolog to Java with InterProlog

On the PROLOG-side, the `javaMessage/2` predicate allows a programmer to send messages to a JAVA object. The example shown in code snippet 2.7 illustrates how to send the message `println` to the static variable `out` in the class `java.lang.System`. The operator `'-'` is employed to refer to the field of an object or a static variable in a class.

```

1 javaMessage('java.lang.System'-out,
2     println(string('Hello World!')))
3 ).
```

Snippet 2.5: The INTERPROLOG `javaMessage/2` predicate.

INTERPROLOG allows to express JAVA objects as “object specifications”. Code snippet 2.6, extracted from [13], illustrates how a JAVA Integer representing the number 13 is expressed as an object specification on the PROLOG-side. Such specification “corresponds to the blueprint for a new object instance on the JAVA-side”. In this way, INTERPROLOG defines an automatic conversion between JAVA objects and PROLOG terms based on the JAVA serialisation API [106] (on the JAVA-side) and a *definite clause grammar* [109] (on the PROLOG-side). In other words, the term representation of an arbitrary JAVA object is derived from its serialisation as a plain stream of bytes. The grammar used by INTERPROLOG allows to obtain a more readable representation, more amenable for reasoning and pattern matching in PROLOG.

```

1 IntegerObject=object(
2     class(java.lang.Integer,long(4834,-24412,-2175,-30920),
```

```

3      classDescInfo([int(value)],2,
4          class(java.lang.Number,long(-31060,-27363,2964,-8053),
5              classDescInfo([],2,null))),
6  [] + [] + [13]);

```

Snippet 2.6: An object specification in INTERPROLOG.

INTERPROLOG allows to start and manipulate PROLOG engines from JAVA programs by means of instances of a class reifying a PROLOG engine. Alternatively, PROLOG engines are also reified on the PROLOG-side: it is possible to obtain a term representing the current PROLOG engine. Therefore, on the PROLOG-side the term representation of the executing PROLOG engine can be passed to the JAVA-side as a method or constructor argument. This facilitates the inversion of the control towards the object-oriented world and the creation of programs that may need to interact with more than one PROLOG engine.

Code snippet 2.7 illustrates an example of this feature. A representation of the current PROLOG engine is obtained by means of the `ipPrologEngine/1` predicate. Afterwards, it is sent as an argument to the constructor of the `HelloWindow` class defined on the JAVA-side. The implementation of this class is shown in the next section.

```

1 ipPrologEngine(Engine), JavaMessage('HelloWindow','HelloWindow'(Engine)).

```

Snippet 2.7: The INTERPROLOG `javaMessage/2` predicate.

From Java to Prolog with InterProlog

Code snippet 2.8 shows the implementation of the `HelloWindow` class introduced in the previous section. As described before, the class expects an object reifying a PROLOG engine in its constructor (line 3).

This example illustrates how INTERPROLOG manages JAVA references on the PROLOG-side. A `JTextField` component is instantiated (line 6) and is explicitly registered as a foreign object in the PROLOG engine (line 7). Lines 8 to 13 provide the code for showing a window containing the previous text field and a button using the Swing library. When the button is pushed, it queries the `greatat/1` predicate, passing as argument the handler to the text field object (line 16).

```

1 public class HelloWindow extends JFrame {
2     PrologEngine myEngine;
3     public HelloWindow(PrologEngine pe) {
4         super("Java-Prolog-Java call example");
5         myEngine = pe;
6         JTextField text = new JTextField(15);
7         final Object fieldObject = myEngine.makeInvisible(text);
8         text.setBorder(BorderFactory.createTitledBorder("text"));
9         JButton button = new JButton("Greet");
10        Box box = new Box(BoxLayout.Y_AXIS);
11        box.add(text); box.add(button);
12        getContentPane().add(box);
13        setSize(200,100); show();
14        button.addActionListener(new ActionListener() {
15            public void actionPerformed(ActionEvent e) {

```

```

16         myEngine.deterministicGoal("greetat(Obj)","[Obj]",new ←
           Object[] {fieldObject});
17     }
18 }));
19 }
20 }

```

Snippet 2.8: From JAVA to PROLOG example with INTERPROLOG.

The definition of the `greetat/1` predicate is shown in code snippet 2.9. The predicate invokes the JAVA method `setText` on the text field reference received as argument. The argument of the `setText` method is the string “Hello World!”, which will be shown in the text field when the `greetat/1` predicate is queried. Note that a JAVA string is expressed with the compound `string/1`.

```

1 greetat(_Text) :- JavaMessage( _Text, setText(string('Hello world!')) )

```

Snippet 2.9: Sending a message to an object reference with INTERPROLOG.

InterProlog Summary

Although INTERPROLOG provides automatic conversions from JAVA objects to PROLOG terms and vice-versa, a programmer has no control over the term representation of such JAVA objects. Instead, a fixed object-specification built from the serialised object data is employed. This technique has the disadvantage that it can only work with objects implementing the JAVA `java.io.Serializable` interface, which limits to a certain extent its applicability. An advantage of the serialisation-based technique of INTERPROLOG, however, is that it works out of the box with objects having circular references (*e.g.*, an object having itself as one of its properties).

An advantage of expressing JAVA Strings by means of the `string/1` compound consists in simple atoms being free to be used to reference JAVA classes, as illustrated by code snippet 2.7. However, an inconvenience of this design decision consists in calls to methods including JAVA Strings as arguments are less straightforward and concise.

From a functionality perspective, a limitation of INTERPROLOG is that it does not support non-deterministic queries. Instead, if a programmer requires more solutions than the first one, she is forced to rely on meta-predicates such as `findall/3` for getting all the solutions to a query. This is quite inefficient in many scenarios (*i.e.*, the query may demand a considerable amount of time to process all solutions when the programmer was interested in only a small subset of them).

2.2.3 PDT Connector

The PDT CONNECTOR library [113] is a plugin component for programming in PROLOG from the ECLIPSE development environment [102]. It can also be used as a standalone JAVA-PROLOG bridge library. Its current implementation

is socket-based and unidirectional, supporting only the access of PROLOG from JAVA programs. At the time of writing, it is compatible with SWI PROLOG and there is ongoing development to make it compatible with YAP PROLOG.

PDT CONNECTOR, having to a certain extent different goals than other JAVA-PROLOG bridge libraries (high-performance tooling support), does not attempt to provide advanced linguistic integration features. It does provide, however, interoperability features offering a fine-grained control over the PROLOG processes from the JAVA-side. For example, multiple PROLOG engines can be easily created, stopped and restarted. It also provides advanced features related to multi-threading, such as asynchronous querying.

As an illustrative example extracted from the PDT CONNECTOR documentation, code snippet 2.10 shows how a PROLOG engine can be queried about the family relations between relatives. First, a PROLOG interface labelled "Example" is created on line 2. If a PROLOG interface with that label already exists, it will be returned instead of instantiating a new engine. On line 6, a textual representation of the compound `father_of(Father, peter)` is created by means of the auxiliary method `bT()`. This compound is the goal of a query executed on line 8. The result is a map of `String` to `Object`, where the keys of the map are the names of unbound variables in the query and the values the string representation of the terms bound to those variables. If there are no answers to the query, the returned value is `null`.

Another query is created on line 22. This time all the results of the query are requested. These are modelled as a list of maps, where each map corresponds to an individual solution. If there are no solutions to the goal, the list is empty. The list of solutions is iterated (lines 23–26) and a textual description of each result is printed on the console (line 25).

```

1 // get Prolog interface ... will be created if it doesn't already exist
2 PrologInterface pif = PrologRuntimeUIPlugin.getDefault().←
    getPrologInterface("Example");
3 try {
4     // create query with the buildTerm method (org.cs3.prolog.common.←
        QueryUtils)
5     // this is the same as "father_of(Father, peter)"
6     String query = bT("father_of", "Father", "peter");
7     // get the first result of the query (ignore other results if there ←
        are any)
8     Map<String, Object> result = pif.queryOnce(query);
9     if (result == null) {
10         // if the result is null, the query failed (no results)
11         System.out.println("peter has no father.");
12     } else {
13         // if the query succeeds, the resulting map contains mappings ←
            from variable name to the binding
14         System.out.println(result.get("Father") + " is the father of ←
            peter.");
15     }
16
17     // create another query
18     query = bT("father_of", "john", "Child");
19     // get ALL results of the query as a list
20     // every element in this list is one result
21     // if the query fails, the list will be empty (but it won't be null)
22     List<Map<String, Object>> results = pif.queryAll(query);
23     for (Map<String, Object> r : results) {

```

```

24         // iterate over every result
25         System.out.println(r.get("Child") + " is a child of john.");
26     }
27 } catch (PrologInterfaceException e) {
28     e.printStackTrace();
29 }

```

Snippet 2.10: From JAVA to PROLOG example with PDT CONNECTOR.

2.2.4 Jasper

JASPER [118] is a JNI-based bi-directional interface between programs written in JAVA and SICSTUS PROLOG [16]. In JASPER, multiple PROLOG engines can be created from the JAVA-side and multiple JAVA virtual machines can be started from the PROLOG-side. In the rest of this section we discuss features related to bidirectional integration using JASPER.

From Prolog to Java with Jasper

JAVA methods can be mapped to PROLOG predicates by means of the `foreign/3` predicate. Method arguments are converted automatically according to the arguments declared in the `foreign/3` predicate. However, this conversion is mainly scoped to JAVA-PROLOG primitive types with certain exceptions (*e.g.*, a term may be converted to the reification of a class on the JAVA-side). Method arguments in JAVA may be objects reifying PROLOG terms. Furthermore, such terms can be PROLOG variables that may be explicitly unified with a PROLOG term in the method execution.

Code snippet 2.13 shows an example of the mapping of the `simple/2` predicate to the `simpleMethod` static method which is shown in code snippet 2.12. On line 7, the predicate is specified as expecting as input an integer as its first argument (the parameter of the JAVA method) and having as output in the second argument another integer (the return value of the JAVA method).

```

1 :- module(simple, [simple/2]).
2
3 :- use_module(library(jasper)).
4 :- load_foreign_resource(simple).
5
6 foreign(method('Simple', 'simpleMethod', [static]), java,
7         simple(+integer,[-integer])).
8
9 foreign_resource(simple,
10                [
11                    method('Simple', 'simpleMethod', [static])
12                ]).

```

Snippet 2.11: Mapping a PROLOG predicate to a JAVA method.

```

1 public class Simple {
2     static int simpleMethod(int value) {
3         return value*42;
4     }
5 }

```

Snippet 2.12: A JAVA class declaring a static method.

Code snippet 2.13 shows an example of how the predicate can be used from PROLOG, once the mapping has been defined.

```

1 | ?- simple(17,X).
2
3 X = 714 ?
4
5 yes
6 | ?-

```

Snippet 2.13: Invoking a JAVA method from PROLOG using JASPER.

JASPER can also manage JAVA references on the PROLOG-side. JAVA objects returned by JAVA methods are represented in PROLOG as '\$java_object'/1 terms. These references are valid until PROLOG returns to JAVA or the reference is explicitly deleted by means of a dedicated predicate. Alternatively, a reference can be made global, ensuring that it will remain valid until explicitly deleted.

From Java to Prolog with Jasper

JASPER allows to execute both deterministic and non-deterministic queries from JAVA. Code snippet 2.14 shows an example, extracted from the SICSTUS documentation, illustrating the interaction with PROLOG from the JAVA-side. A SICSTUS engine is explicitly created (line 13) and a logic theory is loaded in it (line 14). The structured representation of a query is created on line 20. All solutions of the query are traversed and the binding of one of its variables is printed on the console (lines 22-24).

```

1 import jasper.*;
2
3 public class Simple {
4
5     public static void main(String argv[]) {
6         SICStus sp;
7         SPPredicate pred;
8         SPTerm from, to, way;
9         SPQuery query;
10        int i;
11
12        try {
13            sp = new SICStus(argv,null);
14            sp.load("train.ql");
15            pred = new SPPredicate(sp, "connected", 4, "");
16            to = new SPTerm(sp, "Orebro");
17            from = new SPTerm(sp, "Stockholm");
18            way = new SPTerm(sp).putVariable();
19
20            query = sp.openQuery(pred, new SPTerm[] { from, to, way, way ←
21                });
22
23            while (query.nextSolution()) {
24                System.out.println(way.toString());
25            }
26        } catch ( Exception e ) {
27            e.printStackTrace();
28        }
29    }
30 }

```

```

27         }
28     }
29 }

```

Snippet 2.14: From JAVA to PROLOG example with SICStus.

Jasper Summary

After configuring the mapping between predicates and methods, JASPER provides a concise mechanism for invoking JAVA routines from the PROLOG-side. However, one of the drawbacks of this approach is the limited mappings that can be established between types in PROLOG and JAVA. In addition, the amount of required mapping data that should be specified for each method call is considerable. An improved approach should attempt to infer those mappings when possible.

Although the management of JAVA references on the PROLOG-side is mainly explicit, the automatic release of local JAVA references after PROLOG returns control to JAVA could simplify the interoperability code in certain scenarios.

Regarding the API from JAVA to PROLOG, a significant difference that we have found in JASPER as compared to other integration libraries, is that a JAVA object reifying a PROLOG variable is automatically bound to different values during the traversal of the solutions to a query. Other libraries typically reify the concept of a “query solution” as a first class object that provides different variable bindings for each distinct solution.

2.3 Embedded Logic Engines

In this section we discuss engines embedded within an object-oriented program. We have included in our survey engines targeting both statically and dynamically typed object-oriented languages since they provide different integration strategies.

Despite often being less performant than their native counterparts, embedded logic engines typically provide a higher degree of integration from the language perspective. Some advanced integration features present in our work have been inspired or adapted from those engines.

2.3.1 tuProlog

TUPROLOG [44, 110] is a lightweight PROLOG engine embedded in JAVA. It provides the following features [45]:

Minimality: TUPROLOG attempts to be as thin and light-weight as possible. In this way, the library is convenient for devices with few resources.

Dynamic and Open Configurability: Both static and dynamic configuration of components are possible. Nothing prevents to add new add-ons at run-

time, the only requirement being that a component should be in the classpath before being added to a TUPROLOG running application.

Full & Clean Prolog/Java Integration: TUPROLOG attempts to provide simple access to JAVA resources from PROLOG and to avoid complex setup procedures. It also allows to interact straightforwardly with a PROLOG engine from a JAVA program. One of its core objectives is to avoid that the integration mixes the logic and object-oriented paradigms, or alters the nature of either PROLOG or JAVA. This principle is identified as “preserving orthogonality in the paradigm integration”.

TUPROLOG identifies three main use-cases. Two related to the bidirectional communication (From JAVA to PROLOG and from PROLOG to JAVA) and one related to the possibility of using JAVA as the implementation language for writing new PROLOG libraries (referred as “JAVA for PROLOG”).

From Java to Prolog with tuProlog

The interaction from JAVA to PROLOG allows multiple independent engines, each of them potentially configured with different logic theories. The example shown in code snippet 2.15 illustrates how to instantiate a TUPROLOG engine from JAVA.

```

1 Prolog engine = new Prolog();
2 Theory t = new Theory(new java.io.FileInputStream("math.pl"));
3 engine.setTheory(t);
4 SolveInfo answer = engine.solve("dExpr(sin(2*x)*cos(x), Der)");
5 Term derivative = answer.getTerm("Der");
6 Term newGoal = new Struct("evalExpr", derivative, new Double(0.5), new ←
    Var("X"));
7 SolveInfo result = engine.solve(newGoal);
8 double value = ((Number)result.getTerm("X")).getDouble();

```

Snippet 2.15: From JAVA to PROLOG with TUPROLOG.

A PROLOG engine is created on line 1. A new theory is created from a file (line 2) and loaded in the engine (line 3). The solution to a query, whose goal is expressed in a textual form, is found on line 4. A variable binding in the solution is extracted on line 5. The next lines repeat the same steps with another goal expressed in a structured form (line 6).

From Prolog to Java with tuProlog

The scenario of a JAVA library accessed from within PROLOG is illustrated in code snippet 2.16. A JAVA object is instantiated and represented as a PROLOG atom which is bound to the variable `Dialog` (line 1). The operator `<-` allows to invoke a JAVA method on a JAVA object referenced by an atom (line 2). The `returns` operator allows to obtain a handler in PROLOG to the JAVA object returned by a method invocation on the JAVA-side (line 3).

```

1 choose_file(File) :-

```

```

2     java_object('javax.swing.JFileChooser', [], Dialog),
3     Dialog <- showOpenDialog(_),
4     Dialog <- getSelectedFile returns File.

```

Snippet 2.16: From PROLOG to JAVA with TUPROLOG.

TUPROLOG also allows to associate a JAVA object with an arbitrary PROLOG term. This can be done by means of the `java_object/3` predicate. Code snippet 2.17 shows an example of this feature. A new instance of the `java.awt.Point` class is instantiated and associated with the PROLOG atom `p1`. This association exists during the current PROLOG proof. Therefore, when the proof ends, `p1` loses any special meaning since its associating with a JAVA object has gone out of scope. A PROLOG term cannot be assigned to another JAVA reference while its association is still in scope. A programmer can configure if this behaviour is enabled or if conversely an association should survive backtracking by means of setting the `java_object_backtrackable` flag to either `true` or `false`.

```

1  ?- java_object('java.awt.Point', [2,3], p1), p1 <- getX returns X.
2  yes. X / 2.0

```

Snippet 2.17: Object creation with TUPROLOG.

Java for Prolog with tuProlog

TUPROLOG also allows to define PROLOG predicates as libraries implemented in JAVA. Only deterministic predicates can be implemented by means of this technique. Code snippet 2.18 shows the implementation, on the JAVA-side, of the predicate `to_lower_case/2`. The string representation of the first argument is converted into a lower case string (line 3) and explicitly unified with the term sent as second argument (line 4).

```

1  public class StringLibrary extends Library {
2      public boolean to_lower_case_2(Term source, Term dest) {
3          String st = source.toString().toLowerCase();
4          return unify(dest, new Struct(st));
5      }
6  }

```

Snippet 2.18: JAVA for PROLOG in TUPROLOG.

A TUPROLOG predicate implemented as part of a JAVA library can be referenced as any normal PROLOG predicate, once its defining library has been loaded. Code snippet 2.19 shows the loading of the `StringLibrary` (line 1) and the usage of one of its predicates (line 3) as a normal PROLOG predicate.

```

1  ?- load_library('StringLibrary', LibName).
2  yes. LibName / 'StringLibrary'
3  ?- to_lower_case('ABC', LowercaseString).
4  yes. LowercaseString / abc

```

Snippet 2.19: Using a TUPROLOG JAVA library from PROLOG.

tuProlog Summary

In TUPROLOG, the interaction with PROLOG from JAVA is rather explicit and requires the programmer to deal with concepts related to the logic programming side (*e.g.*, PROLOG terms, queries, engines). From the PROLOG-side, access to JAVA objects, although also explicit (*e.g.*, having an explicit `java_object/3` predicate for instantiating an object), has an intuitive and succinct notation thanks to the `<-` and `returns` operators. Furthermore, the automatic management of object references from the object-oriented world may facilitate the integration in many scenarios. Although this behaviour is customisable by means of a configuration flag, a disadvantage may be that a programmer could need to use, in the same program, different strategies for managing JAVA references. Therefore, having a global flag configuring that behaviour may not be the best approach in all circumstances.

In general, the usage of TUPROLOG libraries as described in the previous subsection, provides complete transparency from the logic programming perspective. However, the programmer of the libraries themselves on the JAVA-side has to explicitly refer to logic concepts, such as PROLOG terms. Therefore, the principle of preserving orthogonality in the paradigm integration is not respected with the same degree in all TUPROLOG scenarios.

2.3.2 P@J

P@J [32, 33] is a JAVA–PROLOG integration framework implemented on top of TUPROLOG.

P@J Goal

P@J allows an automatic mapping from JAVA methods to PROLOG predicates using method *type parameters* [101] and *annotations* [91]. This mapping is unidirectional, since P@J is scoped to the JAVA perspective. The main objective of this framework is to allow conciseness in expressing algorithms mixing the two worlds. This goal is identified as reducing as much as possible the *paradigm mismatch* between JAVA and PROLOG.

Reification of Prolog Terms in P@J

P@J makes use of a strongly-typed hierarchy of generic classes reifying PROLOG terms. Code snippet 2.20 shows an extract of this hierarchy [33]. Note that by means of *wildcard types* [76], a term declared as `Term<TermType>` (*e.g.*, `Term<Atom>`) is a super class of both `TermType` (*e.g.*, `Atom`) and `Var<TermType>` (*e.g.*, `Var<Atom>`). This means that a JAVA method expecting as one of its arguments `Term<Atom>` could receive either a concrete PROLOG atom, or a PROLOG variable that could be bound only to an atom.

```

1 class Atom extends Term<Atom>{...}
2 class Int extends Term<Int>{...}
3 class Double extends Term<Double>{...}

```

```

4 class List<X extends Term<?>> extends Term<List<X>>{...}
5 ...
6 class Var<X extends Term<?>> extends Term<X>{...}
7 class Comp<X extends Term<?>> ..

```

Snippet 2.20: P@J term hierarchy.

Symbiotic methods in P@J

In order to provide a transparent mapping between JAVA methods and PROLOG terms, P@J requires a convention ruling the declaration of *symbiotic methods* (*i.e.*, a method automatically mapped to an implementation on the logic side).

This convention dictates that a JAVA method is mapped to a logic predicate having the same name. The predicate arguments are specified as type parameters in the declaration of the JAVA method. Type arguments referring to predicate arguments in PROLOG start with the \$ symbol. In this way, parameters in the JAVA method having one of these type arguments as their type, will be automatically mapped to arguments of the logic predicate. The same applies to the method return value, where its type declaration maps it to a predicate argument. Finally, a method returning a `java.util.iterable<TermType>` is interpreted as returning all solutions to the query, where each individual solution is identified by the type argument of the `Iterable` interface.

But probably P@J conventions are better explained with an example. Code snippet 2.21 shows the declaration of a symbiotic method following such conventions. In line 1, an annotation is used for specifying on the JAVA-side the logic theory required at the execution of the predicate on the PROLOG-side. The method has two type parameters starting with the symbol \$. The only parameter of the JAVA method has as its type the first method type parameter \$X. Hence, this parameter is mapped to the first argument of the predicate in the logic side. The second type parameter \$Y is not used by any of the method parameters. Therefore it is mapped to an unbound variable in the predicate call. However, the return type of the method is the `Iterable` interface which has \$Y as its type argument. Thus, the return value of the method will be an `iterable` which allows traversing all the values bound to the second predicate argument.

```

1 @PrologMethod ( clauses= ... )
2 <$X extends List<Int>,>
3 $Y extends List<Int>> Iterable<$Y> permutation($X l)

```

Snippet 2.21: P@J symbiotic method declaration.

Finally note that given that a symbiotic method is abstract, its declaring class should also be abstract. Therefore, instances of those classes should be obtained by means of a special factory method provided by the framework.

P@J Summary

P@J provides an automatic approach for integrating JAVA and PROLOG. The main shortcoming of this approach is that the types participating in the declaration of symbiotic methods must be explicit PROLOG term types. In other words, there is not an automatic conversion between logic terms, that are natural to PROLOG, to other objects modelling the core business domain. Instead, mappings between such artefacts should be explicitly accomplished every time a symbiotic method is invoked.

An example of this is observable in code snippet 2.21. The argument of the method `permutation` is a list. However, it is not a JAVA list (*i.e.*, an implementation of `java.util.list`) but a special P@J class reifying a PROLOG list. This is because arguments of JAVA methods integrated with PROLOG cannot be arbitrary objects, but instances of classes inheriting from the `Term` class.

2.3.3 Jinni

JINNI [129] is a lightweight, multi-threaded, compiled PROLOG system embedded in JAVA “intended to be used as a flexible scripting tool for glueing together knowledge processing components and JAVA objects in distributed applications” [128].

From Prolog to Java with Jinni

JINNI provides predicates for interacting with classes and instances by means of handlers. For example, the `new_java_class(Name,Cls)` predicate returns a handle to a JAVA class `Cls` with name `Name`. Similarly, `new_java_object(Cls,Args,Obj)` returns `Obj`, an instance of class `Cls`, by calling a constructor with `Args`. Other predicates allow the explicit deletion of object and class handlers from the object table. Handlers can be used to invoke methods on objects or classes (using the `invoke_java_method/5` predicate), or to obtain handlers to instance or static fields (using the `get_java_field/3` predicate) on which various get and set methods can be invoked.

JINNI suggests that “a natural way to interface PROLOG with JAVA is by building PROLOG class wrappers around JAVA classes”. Code snippet 2.24 shows an example of how an instance of the `Hashtable` class is wrapped by means of logic predicates in PROLOG.

```

1 hashtable:-
2   new_java_class('java.util.Hashtable',C),
3   new_java_object(C,void,JavaHashTable),
4   table<=JavaHashTable.
5
6 put_data(Key,Data):-
7   table=>T,
8   invoke_java_method(T,put(Key,Data),_Result).
9
10 get_data(Key,Data):-
11   table=>T,
```

```

12  invoke_java_method(T,get(Key),Data).
13
14  remove_data(Key):-
15      table=>T,
16      invoke_java_method(T,remove(Key),_Result).

```

Snippet 2.22: Wrapping JAVA classes with JINNI.

Once the wrapper predicates have been defined, a PROLOG program can make use of them to implement a concise interaction from the PROLOG-side, as illustrated by code snippet 2.23.

```

1  ?- new(hashtable,H),H:put_data(hello,99),H:get_data(hello,X).
2  H = '$instance'(hashtable@554,1) X = 99 ;

```

Snippet 2.23: Usage of a wrapped class in JINNI.

From Java to Prolog with Jinni

Accessing PROLOG from JAVA is similar to the previous discussed approaches. A fragment of an example extracted from the JINNI documentation is shown in code snippet 2.24. An explicit representation of a PROLOG engine is instantiated on line 1. A query is created and executed from a textual representation on line 2. The first result of the query is shown on line 3.

```

1  Machine M=Top.new_machine();
2  String s=M.run("member(X,[a,b,c])");
3  java.IO.dump("testJinni: first X in member/3: "+s);

```

Snippet 2.24: From JAVA to PROLOG example with JINNI.

Jinni Summary

JINNI provides an explicit API for accessing PROLOG from JAVA and vice versa. Thanks to the JINNI technique for wrapping a JAVA class with a set of logic predicates, a user of the wrapper can abstract away from the JAVA-PROLOG interaction and does not have to think in terms of predicates explicitly referring to JAVA concepts. Particularly, the syntax for invoking JAVA methods from PROLOG is quite intuitive and straightforward. A limitation of JINNI, however, is that for the implementation of the wrapper itself this is not the case, since JAVA concepts need to be made explicit by means of dedicated JAVA interaction predicates. Moreover, a disadvantage of the wrapper approach is that for systems interacting with a considerable number of classes, developing and maintaining such wrappers is costly [138].

2.3.4 Soul

SOUL [143] is a PROLOG-like language embedded in SMALLTALK [60]. It provides a tight symbiotic integration with its host object-oriented language.

Differences with Prolog

Although in general the SOUL syntax is similar to PROLOG, there are some notable differences that are important to highlight before introducing SOUL examples. For example, logic variables are denoted by a question mark, instead of the PROLOG convention of using symbols starting with a capital letter. In addition, lists are delimited by the symbols `<` and `>`. This is due to the fact that the square brackets, used in PROLOG to denote a list, are reserved for delimiting a symbiotic term (*i.e.*, a term wrapping an expression in the host object-oriented language). Also, instead of using the PROLOG `:-` operator for separating the head from its body in a clause, the `if` operator is employed.

Regarding the encapsulation mechanism, SOUL makes use of the notion of *logic layers*, that are somehow equivalent to PROLOG modules.

In the rest of this section we discuss the linguistic integration features of SOUL.

From Soul to Smalltalk

In SOUL, any object belonging to the SMALLTALK host language can be treated as a logic term. Particularly, they can be unified with logic terms or other objects. In addition, the unification of arbitrary objects can be customised by overriding certain methods in the classes of such objects. Since the unification can be customised and defined on a per object basis, this feature has been called *open unification* [10], in the spirit of *open implementations* [79].

Since in SMALLTALK meta-objects are also plain objects, this enables to reflect and reason over the host object-oriented language using logic programming. This technique has been referred to as *inter-language reflection* [67]. Code snippet 2.25 shows a simple example of this by means of the definition of the SOUL meta-predicate `class/1`.

The first clause (lines 1–3) queries for classes in the object-oriented language. Note that the clause fails if the variable `?x` is bound (line 2). Between the square brackets (line 3) there is a SMALLTALK expression. This expression could be arbitrarily complex but in this case just consists of the message `allClasses` sent to the class `System`. This returns a collection of all classes known by the system, what is given as an argument to the `member/2` predicate. The `member` predicate interprets a SMALLTALK collection as a logic list where each member will be bound to the variable `?x` in each solution to the query.

The second clause (lines 5–7) illustrates the case where the variable `?x` is bound (line 6). The logic variable is referenced inside the SMALLTALK block (line 7) as if it were a normal SMALLTALK variable. For example, a SMALLTALK message can be sent to it without any special syntax, or it can be passed as a parameter to a method call. Note that the success of the query depends on the value returned by the `isClass` message. A symbiotic term returning `true` will be considered as a success, and returning `false` as a failure.

```

1 class(?x) if
2   var(?x),

```

```

3      member(?x, [ System allClasses ])
4
5  class(?x) if
6      nonvar(?x),
7      [ ?x isClass ]

```

Snippet 2.25: A logic meta-predicate in SOUL.

SOUL can also automatically map logic predicate calls to SMALLTALK message calls. In SMALLTALK, a keyword message consists of a list of keywords associated with message arguments, as in `obj keyword1: v1 keyword2: v2`. SOUL maps the arguments of a predicate to the keywords arguments. Code snippet 2.26 shows an example. The first condition is evaluated as sending the message `with: 10 with: 20 with: 30` to the `Array` class. Upon evaluation, the variable `?instance` will be bound to a new array with the numbers 10, 20 and 30. The second condition sends the message `at: 2` to this array and binds its returning the value (the number 20) to the logic variable `?value`.

```

1  if Array.with:with:with:(10,20,30, ?instance), ?instance.at:(2,?value)

```

Snippet 2.26: From logic predicates to SMALLTALK messages.

From Smalltalk to Soul

From the SMALLTALK perspective, SOUL provides both a low level API and a symbiotic integration approach. Code snippet 2.27 illustrates the low level querying API. First, an array with binding arguments is instantiated on line 1. This array has arrays of length 2 as its elements. Each of these arrays contains in their first position a symbol indicating a logic variable to be bound and in the second position the value of the binding. An evaluator for a query is created on line 2 passing the binding arguments. These bindings indicate that the variable `?x` should be bound to the object `someClass`. All the results are obtained on line 3. Finally, the solutions for the variable `?y` are queried on line 4.

```

1  argumentArray := Array with: (Array with: #x with: someClass).
2  evaluator := SOULEvaluator eval: 'if hierarchy(?x, ?y)' withArgs: ←
    argumentArray.
3  results := evaluator allResults.
4  ysolutions := results bindingsForVariableNamed: #y.

```

Snippet 2.27: SOUL querying API.

An alternative symbiotic approach attempts to provide an automatic mapping from SMALLTALK messages to queries. This mapping is far from trivial. Gybels identifies the following core issues that need to be tackled in order to solve this problem [66]:

- How to pass unbound logic variables as arguments from the object-oriented side?
- How can a predicate name be derived from a message name?

- What to return if there are many unbound variables in a query?
- How to interpret multiple solutions?
- Which object will the message be sent to?

The first two issues are solved by naming conventions. For example, the SMALLTALK keyword message `add: 1 with: 2 to: 3` will be translated to the SOUL predicate: `add:with:to(1,2,3)`. The message keyword `addwithto: 3` will be translated to the SOUL predicate: `add:with:to(?x,?y,3)`. A limitation of this approach is the difficulty in distinguishing a name implying unbound logic variables from a legitimate, although perhaps unconventional, name (*e.g.*, `addwithto` could also be a valid predicate name in SOUL).

Regarding how to determine the receiver of the message, SOUL proposes two alternative mechanisms.

The first strategy is by means of mapping layers (*i.e.*, logic predicates encapsulation units) to objects stored in global variables. Since “in SMALLTALK classes are also objects stored in global variables, this has the effect of making a predicate-invoking message seem like a message to a class”. For example, the message `Main add: 1 with: 2 to: 3` is translated as the query `if Main.add:with:to:(1,2,3)`.

An alternative syntax does not require the explicit specification of layers in SOUL. This is done by means of the `=` operator used to denote the return value (in the SMALLTALK side) of a SOUL predicate. Code snippet 2.28 shows an example of this. In this case, the predicate returns the number 10 (line 1) if it succeeds.

```

1 ?product discountFor: ?customer = 10 if
2   ?customer loyaltyRating = ?rating &
3   ?rating isHighRating

```

Snippet 2.28: Specifying the return value of a logic clause.

Behind the curtains, this is implemented by means of delegating to the SOUL evaluator any message that cannot be understood on the SMALLTALK side. In the example, this occurs if the keyword message `discountFor: customer` is not understood in SMALLTALK.

Explicitly specifying the return value of a SOUL clause solves the problem of what value to return in case many unbound variables are present. This leaves one remaining issue concerning how to deal with predicates having more than one solution. In certain cases a programmer may want to obtain only one solution and in others a collection with all the solutions. SOUL solves this by considering a query solution always as a collection. In case a query has only one solution, any message that is not understood by the collection object is delegated to the first element of the collection. In this way, SOUL attempts to reach an “automatic adaptation” between deterministic and not deterministic queries. A disadvantage of this strategy is, however, that subtle ambiguity problems will appear if the first solution to the query is in fact a SMALLTALK collection.

Java compatibility

SOUL is also compatible with JAVA by means of an ECLIPSE plugin [114]. This approach still uses SMALLTALK as the SOUL execution environment, but it interacts behind the curtains with a headless ECLIPSE instance by means of the SMALLTALK native interface.

A new incarnation of SOUL, called EKEKO [40], is embedded in the ECLIPSE environment. It runs as a logic MINIKANREN library [12] embedded in CLOJURE [136]. It allows to analyse and reason using logic techniques not only about JAVA source code, but about the entire ECLIPSE workspace.

Soul Summary

A significative advantage of the SOUL approach is that an explicit causal connection between a SMALLTALK object and its term representation is maintained. This facilitates the further processing of query results, since real SMALLTALK objects (not just their term representation) are bound to logic variables. However, a limitation of this technique is that an arbitrary object always has a unique unification strategy (i.e., it is not context-dependent), instead of allowing to vary its unification according to the requirements of a specific problem. Although SMALLTALK could allow for the dynamic (re)definition of methods (in this case the methods defining the unification strategy), SOUL currently does not provide a simple high-level mechanism to change the logic reification of an object according to its particular context of use.

Limitations and open questions regarding the symbiotic integration from the object-oriented language perspective have been discussed by Gybels [66]. Other SOUL integration features, particularly related to production rules and forward chaining, have been presented by D'Hondt [46].

2.4 Object-Oriented Libraries

Another category of integration approaches concerns object-oriented libraries for logic languages.

These libraries do not define a (multi-paradigm) programming language in the traditional sense. Object constructs do not have a direct textual representation. Instead, the interface to the logic language defines what they look like from the programmer point of view [139].

The lack of a direct textual representation of object-oriented constructs in the logic language is one of the main differences with the multi-paradigm programming language approach discussed in section 2.6. As a representative example, in the rest of this section we discuss the XPCE library [138] and its interface for PROLOG.

XPCE

XPCE/PROLOG is “a hybrid environment integrating logic programming and object-oriented programming for Graphical User Interfaces” [139].

Although the main focus of XPCE is the integration of PROLOG with user interfaces built in an object-oriented language, it is general enough to be used in other use cases.

Integration from the Prolog Perspective

XPCE provides logic predicates for creating references in the object-oriented world (**new/2**), sending a method to an object returning either no value or a boolean success/failure indication (**send/2**), sending a method to an object and capturing its return value (**get/3**) or freeing the memory pointed to by a reference (**free/1**).

XPCE automatically converts to artefacts in the object-oriented language the term arguments of certain predicates (*e.g.*, **new/2**, **send/2** and **get/3**). As in other approaches, it performs an automatic conversion between primitives types. In addition, it considers a compound term as the specification of an instance in the object-oriented world, where the compound name corresponds to the name of the instance class, and the compound arguments to the constructor arguments.

XPCE also performs certain automatic type-based conversions. In code snippet 2.29, a browser is opened (line 1) and its method **members** is invoked (line 2). The method takes an XPCE collection (chain) as argument. Then, the Prolog list **[aap, noot, mies]** is automatically converted into a chain.

```
1 send(new(Browser, browser), open),
2 send(Browser, members, [aap, noot, mies]).
```

Snippet 2.29: Type based conversions in XPCE.

In addition, XPCE allows to pass plain PROLOG terms (*i.e.*, without converting them to foreign objects) as arguments for routines in the object-oriented world. This is enabled when the parameter type in the foreign routine is of the special type **prolog**. If the term argument is not ground, it can be further initialised in the object-oriented world.

XPCE also allows for the definition of classes in PROLOG. These classes can be used to instantiate objects as any other class defined on the object-oriented side. A small example is shown in code snippet 2.30. In this example, a new class **my_box**, inheriting from the built-in class **box**, is defined. The class overrides the implementation of the **event** method (line 3–9) to fill in the colour of the box with red (line 5) when the mouse cursor enters its area (line 4) and cleans its colour (line 7) when the cursor exits (line 6).

```
1 :- pce_begin_class(my_box, box).
2
3 event(Box, Event:event) :->
4     ( send(Event, is_a, area_enter)
5       -> send(Box, fill_pattern, colour(red))
```

```

6      ; send(Event, is_a, area_exit)
7      -> send(Box, fill_pattern, @nil)
8      ; send_super(Box, event, Event)
9      ).
10
11 :- pce_end_class(my_box).
```

Snippet 2.30: Defining a class in XPCE.

Given that XPCE allows the definition and modification of object-oriented classes from the logic language side, this facilitates the extensibility and customisability of a system making use of this library.

Integration from the XPCE Perspective

From the XPCE perspective, a singleton instance of a class `Prolog` allows to interact with the PROLOG side. This object provides a method taking a predicate name and a list of arguments as an input. The object can also be activated from the PROLOG side, as illustrated in code snippet 2.31, where it is referred with the term `@prolog`.

```

1 ?- send(@prolog, writeln('Hello World')).
2
3 Hello World.
```

Snippet 2.31: Activating the `Prolog` object in XPCE.

A limitation of XPCE from the object-oriented language perspective is the lack of support for non-determinism and for expressing complex queries (*i.e.*, queries composed from more than one single predicate).

Embedding of Multiple Host Languages in XPCE

Although XPCE is single threaded, it is possible to embed it in multiple host languages. For example, QUINTUS [83] and SWI PROLOG can be connected to the same XPCE process by means of `send(@quintus, ...)` from SWI and vice versa.

2.5 Other Integration Approaches

In addition to the approaches surveyed before in this section, there are other integration libraries targeting similar languages and whose features are subsumed by previous visited techniques. Examples include JAVA, SCALA, and PYTHON implementations of PROLOG [31, 7, 58, 131] or other libraries for integrating native PROLOG engines with an object oriented program (*e.g.*, the bidirectional CIAO JAVA INTERFACE [11, 73]) or native libraries (*e.g.*, WIN-PROLOG). Several of these techniques are limited to an integration from the logic language perspective, mainly by offering a set of built-in PROLOG predicates that enable easy access to the object-oriented language.

In many cases, libraries for calling routines from the object-oriented language to the logic world are provided, but they fail to abstract the programmer from low-level mappings, requiring an explicit representation of logic concepts (logic engine, queries, logic terms) in the object-oriented program. The same problem occurs for rule engines embedded in JAVA (*e.g.*, JESS [57]) that use a declarative language other than PROLOG.

An approach integrating JAVA with a logic constraint solver is presented by Majchrzak et al. [84]. This work relies on a symbolic virtual machine and the syntax of JAVA programs is left unmodified. Methods evaluated as logic computations are identified with an annotation. Logic variables are also identified with annotations and are limited to JAVA primitive types. A limitation is the lack of adaptation of the result of a logic method, as in the SOUL approach; instead all logic methods must return an object instance of the special class `Solutions`.

In another category, there are techniques that integrate the two paradigms extending the syntax of the object-oriented language to include constructs and concepts of the logic language. Some of them focus on allowing the embedding of PROLOG code into JAVA programs, but do not provide any automatic mapping between JAVA objects and logic terms (*e.g.*, PROLOG lists are declared with a special syntax and are not the same as JAVA lists) [92]. We think these approaches converge more to the definition of a new multi-paradigm hybrid language, as explained in section 2.6, supporting the two paradigms than to the implementation of a transparent and automatic linguistic integration.

2.5.1 Translators

As opposed to most previous approaches, translators do not attempt to make interoperate two different programs at runtime. Instead, they transform one program into the other. Examples are PROLOG CAFE [3] and JPROLOG [43], which can convert a PROLOG program into JAVA code. Although translators have interesting applications (*e.g.*, reusing legacy logic-programming code in an environment where no logic engine is available), we have left those approaches out of our research scope.

2.5.2 Code Generation, Querying and Transformation Libraries

We found several tools providing support for generating, querying and transforming object-oriented code from within a logic language [41, 42, 117]. Most of them have similar features to SOUL, but do not attempt to provide integration support from the object-oriented language perspective. Furthermore, the features from the logic language perspective are focused on meta-programming but are difficult to use for something else (*e.g.*, to symbiotically call object-oriented routines from the logic side).

There are also generative approaches making use of quasiquotation that are employed to generate foreign language code, not necessarily object-oriented, from a logic program [141].

2.6 Multi-Paradigm Languages

A multi-paradigm language combines concepts from different programming paradigms in one single language. This has the advantage that no integration approach is required between distinct languages, since concepts belonging to different paradigms coexist in the same language.

An example of an object-oriented language providing support for logic programming features is OZ [115].

Examples of logic languages also supporting object-oriented features are LOGTALK [95, 94], FLORA [145], SICSTUS OBJECTS [52] and O’CIAO [112].

In the rest of this section we give an overview of LOGTALK, which can be considered as a representative example among the existing approaches. LOGTALK is also used as one of the underlying building blocks of our approach. Therefore, it is referenced in several chapters of this dissertation.

Logtalk

LOGTALK supports both prototype-based or class-based hierarchies [127, 6], even in the same application. The first-order LOGTALK entities are *objects*, *protocols* and *categories*. Since we make use of certain LOGTALK artefacts to implement the components of our approach, we provide a short overview of them in this section.

Logtalk Objects

Objects serve as encapsulation units for predicates, thus providing namespaces allowing the separation of code in manageable units.

In LOGTALK, the term *object* is used in a broad sense to denote a *prototype*, *parent*, *class*, *subclass*, *superclass*, *metaclass* and an *instance*.

Object names can be atoms or compound terms. A LOGTALK object is textually defined by means of the directives: `object/1-5` and `end_object/0`.

Code snippet 2.32 illustrates a declaration of an object named `hello_object`. The `public/1` directive on line 3 declares the `greet/0` predicate as part of the object’s public interface (*i.e.*, it can be called from any object). Other possible access modifier directives are `private/1` (*i.e.*, it can be called only from the container object) and `protected/1` (*i.e.*, it can be called by the container object or by the container’s descendants). The implementation of the predicate `greet/0` (line 4) just shows on the screen the 'Hello World' message.

```

1 :- object(hello_object).
2
3     :- public(greet/0).
4     greet :- write('Hello World').
5
6 :- end_object.
```

Snippet 2.32: Declaring a LOGTALK object.

Sending Messages to Objects

In LOGTALK, a message to an object can be sent using the `::/2` operator. Therefore, we could write `hello_object::greet` to send the message `greet` to the object `hello_object`.

If the message is not declared for the object an exception will be thrown. If the message is declared, but not defined, the message will simply fail (*closed world assumption* [55]).

Parametric Objects

When an object name is a compound term, it is referred to as a *parametric object* [96]. Code snippet 2.33 shows an alternative implementation of the previous object this time using the compound `hello_object/1` as its name. The implementation of the `greet/1` predicate (lines 4–7) refers to the first parameter of the object by means of the `parameter/2` predicate (line 6). Afterwards this parameter is shown on the screen (line 7).

```

1 :- object(hello_object(_Name)).
2
3     :- public(greet/0).
4     greet :-
5         write('Hello '),
6         parameter(1, Name),
7         write(Name).
8
9 :- end_object.

```

Snippet 2.33: Declaring a LOGTALK parametric object.

Prototypes

Prototypes are either self-defined objects or objects that are defined as extensions to other objects, their parent prototypes. With prototypes, there is no distinction between abstractions and concrete examples of these abstractions, as we have with classes and instances. Therefore, prototype semantics is simpler as there is only one possible relation between objects (extension), while with classes and instances we have to deal with both instantiation and specialisation relations.

For example, the `hello_object/1` parametric object shown in code snippet 2.33 can play the role of a prototype. Then, an object `talkative_hello_object/1` can extend it as shown in code snippet 2.34, line 2. In this example, the `greet/0` predicate first delegates to its inherited definition (line 5) and then shows additional text on the screen (line 6).

```

1 :- object(talkative_hello_object(_Name),
2     extends(hello_object(_Name))).
3
4     greet :-
5         ^^greet,
6         write('. How are you ?').
7

```

```
8 :- end_object.
```

Snippet 2.34: Declaring a LOGTALK parametric object.

Classes

A class is a role played by a LOGTALK object. In other words, there is no `class` keyword specifying that an object is a class. Instead, an object is considered a class or an instance if it, respectively, specialises (code snippet 2.35) or instantiates (code snippet 2.36) another object (that would be playing the role of a class).

```
1 :- object(object,
2       specializes(class)).
3
4       % predicates common to all instances of the class
5
6 :- end_object.
```

Snippet 2.35: Object specializing a class.

```
1 :- object(object,
2       instantiates(class)).
3
4       % predicates common to all instances of the class
5
6 :- end_object.
```

Snippet 2.36: Object instantiating a class.

If an object plays the role of a class, it declares and possibly defines predicates for its instances.

Protocols

Protocols enable separation between interface and implementation. They are equivalent to the notion of interfaces in languages such as JAVA.

A protocol can only contain predicate declarations concerning a single functionality. In LOGTALK, a protocol can be implemented by several objects and an object can implement several protocols. A LOGTALK protocol is defined using of the directives `protocol/1-2` and `end_protocol/0`, as demonstrated in code snippet 2.37.

```
1 :- protocol(hello_protocol).
2
3       :- public(greet/0).
4
5 :- end_protocol.
```

Snippet 2.37: Declaring a LOGTALK Protocol.

Both objects and categories can implement a protocol, as shown in code snippets 2.38 and 2.39.

```

1 :- object(Object,
2     implements(hello_protocol)).
3
4     greet :- ...
5     ...
6 :- end_object.

```

Snippet 2.38: An object implementing a LOGTALK protocol.

```

1 :- category(Object,
2     implements(hello_protocol)).
3
4     greet :- ...
5     ...
6 :- end_object.

```

Snippet 2.39: A category implementing a LOGTALK protocol.

When a protocol is implemented by an object (or a category; described next), the definition of all its declared methods is not mandatory. When an object receives as a message a predicate declared by a protocol where such predicate has been left undefined the message will fail. This is because LOGTALK implements a closed world assumption, where the absence of a definition implies failure. This is contrary to an object receiving an unrecognised message, where in this case an error will be thrown.

Categories

LOGTALK also supports categories, which are fine-grained units of code reuse and can be seen as a dual concept of protocols that can contain both predicate declarations and definitions. As protocols, categories should describe a single functionality. The concept of categories is similar to the concept of **mixins** [8] and “provide a way to encapsulate a set of related predicate declarations and definitions that do not represent an object and that only make sense when composed with other predicates” [98]. Categories can be (virtually) imported by any object, independently of the roles that object plays (*e.g.*, prototypes or classes). A category cannot be used as a stand-alone object, *i.e.*, it cannot be the target of a LOGTALK method invocation.

An example of a simple category declaration is shown in code snippet 2.40. As the example shows, a category is defined using the directives **category/1-2** and **end_category/0**.

```

1 :- category(hello_category).
2
3     :- public(greet/0).
4     greet :-
5         parameter(1, Name),
6         write(Name).
7
8 :- end_category.

```

Snippet 2.40: Declaring a LOGTALK category.

The syntax for importing a category into an object is shown in code snippet 2.41.

```

1 :- object(Object,
2       imports(hello_category)).
3     ...
4 :- end_object.
```

Snippet 2.41: An object importing a LOGTALK category.

Logtalk Portability

One advantage of LOGTALK in comparison with other alternatives is its wide portability across PROLOG implementations. For a description of LOGTALK other features we refer to the LOGTALK documentation [98, 97, 93].

2.7 Synthesis of the State of the Art

In this section we synthesise the integration features that we have found in existing approaches. We have purposely not included domain-specific features such as querying, generating and transforming object-oriented code from the logic language. Although these are interesting features with concrete applications (*e.g.*, building advanced code development tools [40, 80, 25]), our interest is in general integration techniques that can be applied to most hybrid logic and object-oriented systems.

Not all features presented in this section are relevant to all possible hybrid architectures. For example, in logic engines embedded in an object-oriented program it may not make sense to explicitly reify the object-oriented environment itself or developing special techniques for dealing with the life cycle of references from the object-oriented world. However, since our goal is developing a portable approach independent of a particular architecture, we have taken into consideration all those properties.

After a description of each integration feature, in the next section we present a comparative table showing which of them are available in which of the surveyed approaches, thus establishing a “design space” of relevant features and approaches.

2.7.1 Integration Degree

This feature concerns the obliviousness and coverage of an integration approach.

Obliviousness

This property measures how explicit or implicit an integration approach is. It ranges from libraries allowing an explicit manual integration (*e.g.*, JPL) to linguistic approaches that attempt a transparent and automatic integration (*e.g.*, SOUL). While the former typically causes the integration code to be

tangled with the main concern of the application, the latter attempts to make both the logic and object-oriented language oblivious of being integrated with one another [46]. This facilitates a proper *separation of concerns* facilitating the development and maintenance of the system [48, 107].

Obliviousness of Surveyed Approaches

In comparison to the level of integration reached by logic engines non-embedded in the object-oriented environment, approaches targeting embedded logic engines provide, in general, higher level integration features. To a certain extent, this is explained because sharing the same memory space makes easier certain interoperability problems that present a major complexity when interacting with a non-embedded logic engine. Although interacting with non-embedded logic engines is arguably a more common real business scenario, their integration API typically makes use of explicit inter-language calls.

From most surveyed approaches we found a common limitation the lack of support for (semi-)automatically converting complex objects (*i.e.*, non-primitive types) to an adequate term representation. In that sense, XPCE and, to a certain extent, INTERPROLOG are the only approaches that attempt to do something in that direction. XPCE automatically converts to objects compound terms which are arguments in method calls. It assumes that the compound name is the name of an object class and the compound arguments the constructor arguments. INTERPROLOG provides an automatic generation of the term representation of an object based on its serialised stream of bytes.

Another observed problem was the lack of support for evaluating foreign routines or expressions in a way that does not make explicit the transition to a different language. Although in most approaches it is possible to write ad hoc wrappers that help to accomplish a transparent transition, the wrappers themselves have to be manually written and many low-level mapping operations should be explicit in their implementation.

Regarding embedded logic engines, although their integration is comparatively quite superior to other kind of engines, there is still room for improvement. For example, besides trivial conversions between primitive types, most approaches miss support for automatic conversion of inter-language artefacts. SOUL is a notable exception, since objects can transparently pass from the object-oriented to the logic environment and vice versa. This is because the SOUL approach does not require converting an object to a term representation. Instead, objects can be unified with logic terms or other objects on the basis of a unification mechanism customisable by means of implementing a certain method. One shortcoming of the SOUL approach is related to portability since it requires a logic engine embedded in the object-oriented environment. Other problems and limitations related the obliviousness of the integration, specially from the object-oriented perspective, have been discussed by Gybels [66].

Coverage

The coverage of the approach concerns the two possible integration directions. We define a *bidirectional integration* as an integration that is accomplished both from the object-oriented and logic environment. A *unidirectional integration* is focused on one single direction: either accessing the logic world from the object-oriented side or vice versa.

Coverage of Surveyed Approaches

Most of the surveyed approaches are bidirectional. An exception is PDT CONNECTOR, targeting only the integration from the object-oriented language perspective. P@J inherits from TUPROLOG its bidirectional integration features.

2.7.2 State Sharing

This feature describes how foreign references are managed.

Accessing References from the Object-Oriented Language in the Logic Language

In the object-oriented world, objects often refer to a store location in memory that keeps their state. These references are typically allocated at instantiation time and, depending on the language, explicitly released when not required anymore (*e.g.*, C++ [124]) or automatically scheduled for garbage collection (*e.g.*, JAVA [62]).

Therefore, from the logic language perspective, foreign reference management is concerned about how to deal with the life cycle of references in the object-oriented world. This typically determines whether the foreign reference life cycle (*e.g.*, the life span of the reference) should be manually configured or an automatic mechanism is available.

Many approaches provide a simple mechanism for accessing the state of objects from the logic side. In several cases, complex objects can only be referred by opaque handlers (*e.g.*, JPL) where the internal state cannot be examined from PROLOG. Other approaches allow to reify the full object state on the logic side, sometimes at the cost of losing the identity of the object (*e.g.*, INTERPROLOG allows to generate an automatic representation of an object based on its serialised stream of bytes and a definite clause grammar).

It is difficult to find a meaningful representation that suits the logic programmer in any scenario. We claim that the programmer should be able to decide on such a representation and that this is an orthogonal property to the preservation of the object identity.

Accessing References from the Logic Language in the Object-Oriented Language

The notion of state does not exist in most logic languages. For example, in PROLOG “a logic variable stands for an unspecified but single entity rather than for a store location in memory” [123].

From the object-oriented language perspective, manipulating references typically refers to being able to pass a non-unified logic variable as an argument to a routine in the object-oriented world. The variable should have a convenient object-oriented representation. Furthermore, a semantics for unifying it with another term, either implicitly or explicitly, should exist. The value unified to the variable could be any data type understood by the logic language and the unification should be consistent with existing unification rules on the logic side. For example, if the same variable is passed more than once as an argument to a method, unifying one of them should also affect the other.

State Sharing in the Surveyed Approaches

Most approaches supporting the interaction with the object-oriented environment from the logic world allow some mechanism for accessing and mutating the state of objects. An exception is INTERPROLOG, where state of opaque references cannot be queried or modified. However, terms representing an INTERPROLOG object specification expose object state as part of their internal structure.

From the approaches allowing to interact with the object-oriented side from the logic world, most of them support sending logic variables as method arguments to be unified in the object-oriented world. Exceptions are SOUL and INTERPROLOG. What most libraries do to support this is to make use of special objects reifying a logic variable as the arguments of methods on the object-oriented side. The library typically provides routines allowing the explicit unification of these logic variables with a logic term.

2.7.3 Behavioural Integration

This integration feature concerns the possibility of invoking foreign routines or invoking foreign expressions of arbitrary complexity.

Invocation of Methods from the Logic Language

From the logic perspective, this property typically requires determining the receiver of the invocation (*e.g.*, an object for instance methods or a class for class or static methods) and the name and arguments of the method. Some of the arguments of the routine may not be ground, so a semantics for interpreting unbound variables in the object-oriented world should be defined. Also, a mechanism for capturing the return value of a method on the logic side and interpreting it as a logic term should be established.

Querying Predicates from the Object-Oriented Language

From the object-oriented language, this implies querying predicates at the logic side and defining how to interpret a query result as an object.

Evaluation of Foreign Expressions

Invoking foreign routines is a special case of the more general problem of evaluating foreign language expressions and interpreting its outcome in the local language.

From the logic perspective, this implies empowering the programmer to write arbitrarily complex expressions (*e.g.*, a cascade or broadcasting of messages to one or more objects) to be evaluated in the object-oriented environment. From the object-oriented perspective, this implies writing queries that can contain more than one predicate.

Foreign language expressions can be written either by means of a structured representation in the local language or directly in the foreign language. If the foreign language approach is used, the expression can be written either as a text expression to be parsed and executed in the foreign environment or embedded in the local language with some sort of escaping mechanism.

Non-Determinism Support

An object-oriented program interacting with a logic language should deal with non-determinism. This is due to the fact that logic queries may have more than one solution.

This feature presents a particular complexity for approaches attempting to provide an oblivious integration from the object-oriented perspective. This is because the concept of non-determinism exists on the logic side, but does not in the object-oriented environment.

Behavioural Integration in the Surveyed Approaches

All the approaches supporting the interaction with an object-oriented environment support the invocation of methods. The inverse is also true: all the approaches supporting the interaction with a logic engine from the object-oriented world support querying of predicates. Therefore, as it is the only property present everywhere, we can conclude that this is a fundamental feature in any integration library.

However, not all the libraries implement this feature with the same degree of completeness. For example, INTERPROLOG does not support querying non-deterministic goals in the logic side since the available API supports the querying of only the first solution of a goal.

Most approaches supporting the querying of a logic engine also allow for either a textual or structured representation (*i.e.*, using classes reifying logic terms) of a goal. An exception is PDT CONNECTOR, where goals can only be expressed using their textual representation.

Regarding the evaluation of foreign expressions in the local language: from the object-oriented perspective, most approaches allowing to query a single predicate also allow to query a complex goal composed of multiple predicates. From the logic perspective the options were more limited, where only SOUL provided full support for evaluating expressions from the object-oriented world, even written in their original textual representation. XPCE provided some limited support for this feature, allowing to include constructor call expressions as arguments in the invocation of a method or constructor.

2.7.4 Multiplicity of Environments

This feature concerns the number of logic engines that can be interacted with from the object-oriented side and the number of object-oriented environments accessible from the logic side.

Multiplicity of Environments in the Surveyed Approaches

Most of the approaches supporting the integration from the object-oriented perspective allow to create more than one logic engine. Exceptions are JPL and P@J. Although implemented on top of TUPROLOG, which does support multiple PROLOG engines, P@J does not provide mechanisms for targeting the execution of symbiotic methods to a particular PROLOG engine.

From the logic language perspective, this property is only applicable to non-embedded logic engines, since embedded engines normally have an implicit access to the unique object-oriented environment containing them. In our survey we found that only JASPER supports multiple object-oriented environments by instantiating multiple JAVA Virtual Machines.

2.7.5 Customisability

This feature has to do with to which extent it is possible to customise an integration approach.

Customisability in the Surveyed Approaches

Several approaches provide some sort of customisability for the integration. In some cases these are global customisation mechanisms that affect the entire execution environment and are not possible or difficult to change at runtime. Some examples are described below:

- TUPROLOG allows to customise the life span of a relation between a PROLOG term and an object reference. However, this is done by means of a global flag that affects the entire execution environment instead of a particular module or routine.
- P@J allows to customise the mapping of JAVA methods to PROLOG predicates by means of annotations.

- XPCE allows to create and modify at runtime new object-oriented classes from the logic side.
- SOUL allows to customise the unification of arbitrary objects by overriding a special method in those objects.
- JASPER allows to customise, by means of certain directives, how a predicate is mapped to a method.
- PDT CONNECTOR allows to customise a PROLOG engine by means of flags. For example, an engine can be customised to answer the solutions to a query with their textual representation or by means of a structured representation. An advantage of the PDT CONNECTOR approach is that it can be customised at the PROLOG engine level, without a global system configuration.

Our position is that a programmer should be empowered to customise the different aspects of the integration. Furthermore, this customisation should be fine-grained, scoped to different modules, routines or even context-dependent scenarios in a hybrid system.

2.7.6 Portability

Our last integration feature concerns portability. It defines how feasible it is to reuse an integration technique to other implementations of the integrated languages. This property is only applicable when these integrated languages are somehow standardised (*e.g.*, JAVA and PROLOG).

Portability in the Surveyed Approaches

In general, logic engines embedded in object-oriented environments have a low portability. This is because they reach a high integration at the cost of a high coupling with a concrete logic engine that often does not follow standard and de facto conventions. Furthermore, often there is no real distinction between the integration approach and the logic engine itself (*e.g.*, SOUL, TUPROLOG). As a result, it is very difficult to reuse one of these integration techniques without accomplishing a full reimplementation for another logic engine.

Techniques targeting non-embedded logic engines follow a different approach. They are often compatible with more than one logic engine (*e.g.*, JPL is compatible with both SWI and YAP PROLOG) or there is active ongoing work in that direction (*e.g.*, PDT CONNECTOR). There are other libraries that have been compatible with more than one PROLOG implementation in the past, but that currently support only one engine (*e.g.*, XPCE and INTERPROLOG).

We claim that there is a need for an approach providing the high integration observed in embedded logic engines, with the portability of libraries targeting non-embedded ones. In this way, the same integration work can be reused across distinct logic engines.

2.8 Comparison of Integration Features

The integration features presented in this section have been summarised in table 2.1. Below we highlight the most relevant observations extracted from this comparison.

2.8.1 Conventions

In our table, black colour cells indicate that a feature is in general provided. Grey colour cells indicate that the feature is available in a limited way. Finally, a blank cell should be interpreted as the feature being either not provided or with a too superficial implementation.

2.8.2 Interpretation

This table allows to obtain a general view of the strengths and weaknesses of the surveyed tools. However, it has to be evaluated with care since several of the presented tools have a different architecture (*e.g.*, embedded vs. non-embedded logic engines) and goals. Hence, certain integration features are more relevant to them than others.

An intrinsic difficulty about creating such a comparative table lies in the trade-offs between a relative vs. an absolute comparison of techniques. We have followed a relative comparison approach. For example, it can be argued that P@J does provide an oblivious integration from the object-oriented perspective. However, the SOUL approach seems to be superior in that regard. As described in section 2.3.4, SOUL does not require object-oriented methods called from the logic side to have logic terms as arguments, as it is the case with P@J. Therefore, we have drawn that feature for P@J as a grey rectangle and for SOUL we have drawn a black box in the corresponding place. This should not be interpreted as SOUL providing a perfect oblivious bidirectional integration (in fact several limitations are highlighted in [66]), but that after a comparative analysis we found that this particular feature is more advanced in SOUL than in any of the other surveyed techniques.

2.8.3 Relevance of Features

The table allows to extract the most important integration features by observing the frequency those features are implemented by existing tools. For example, with only one exception (PDT CONNECTOR) all the approaches provide a bidirectional integration. Also, with only one exception (XPCE) all tools providing an integration from the object-oriented language allow to query goals of arbitrary complexity. On the other hand, tools providing an integration from the logic language perspective allow at least to execute methods in the object-oriented environment.

There are also features that are rarely provided, like interacting with multiple object-oriented environments from a logic language, as allowed by JASPER.

		Completeness	Obliviousness	State Sharing	Behavioural Integration	Multiple environments	Orthogonal Qualities
		OO to Logic Language (2.7.1) Logic Language to OO (2.7.1)	Obliviousness from the Logic Language(2.7.1) Obliviousness from the OO Language(2.7.1)	Sharing Unbound Logic Variables (2.7.2) Sharing Object State (2.7.2)	Invoking Methods from the Logic Language (2.7.3) Querying Predicates from the OO Language (2.7.3) Evaluating OO Expressions from the Logic Language (2.7.3) Executing Complex Queries from the OO Language (2.7.3) Non-determinism Support(2.7.3)	Multiple Logic Engines (2.7.4) Multiple OO Environments (2.7.4)	Portability (2.7.6) Customisability (2.7.5)
Non Embedded Logic Engines							
JPL							
INTERPROLOG							
PDT CONNECTOR							
JASPER							
Embedded Logic Engines							
TUProlog							
P@J + TUProlog							
JINNI							
SOUL							
Object-Oriented Libraries							
XPCE							

Table 2.1: Comparative Table of Integration Features.

2.8.4 Dependency Correlation

Certain features depend on others to be enabled. For example, the reason no embedded logic engine can interact with more than one object-oriented environment is because many engines do not require such a feature, not because it is a limitation for them.

In addition, an approach not providing integration support from the logic perspective (*e.g.*, PDT CONNECTOR) will not include features that are only relevant from that perspective, such as invoking methods or evaluating object-oriented expressions. This is because such a problem does not exist in the scope of the approach.

2.8.5 Other Correlations

We discovered that some integration features, although not presenting an explicit dependency among them, are strongly correlated. For example, almost all approaches supporting the integration from the object-oriented perspective provided support for non-deterministic queries. Also, most approaches supporting the querying of simple predicates also support the querying of complex goals (*e.g.*, conjunctions of many predicates). Note that the contrary is not true: approaches supporting the execution of a method from the logic side rarely supported the execution of complex expressions from the object-oriented language embedded in the logic language.

2.9 Chapter Summary

In this section we presented a survey of the state of the art in integration techniques for object-oriented and logic languages. Although we are mainly interested in statically-typed object-oriented languages, we have enlarged our survey to also include techniques targeting dynamically-typed languages.

We identified the most important features and concepts provided by existing tools and provided a common vocabulary for referring to them. We then discussed each of those features and described how existing techniques support them.

At the end we summarised all integration approaches and their features, classified them according to the architecture of the approach (*e.g.*, embedded vs. non-embedded logic engines) and presented a comparative table summarising our findings.

Starting from the integration features identified in this chapter, in chapter 4 we will provide a conceptual model of an integration framework for a statically-typed object-oriented language and a logic language. Portability being one of our main objectives, we attempt to improve and make available the advanced integration features observed in embedded logic engines to non-embedded engines implementing a more complex – but common – architecture.

Concrete examples illustrating typical integration scenarios and motivating the need of improved integration techniques are discussed in the next chapter.

3 Case Studies

Few things are harder to put up with
than the annoyance of a good example.
—Mark Twain

In this chapter we illustrate our problem by means of the description of two case studies that can be conveniently implemented as hybrid applications. The first one is focused on integrating an existing PROLOG program within JAVA, while the second one requires the interaction with a JAVA user interface within a PROLOG program. One of our main goals is to illustrate the integration complexity from these two distinct perspectives.

Although the problems discussed here are referenced in the rest of this dissertation, we defer providing a transparent and (semi-)automatic solution to them until chapter 9.

3.1 The London Underground

This case is inspired by a typical problem that can be implemented easily with a logic programming language: a querying system about subway lines and stations in a big city (*e.g.*, to query the number of intermediate stations from one station to another). At the same time, most public transport systems require a user-friendly interface. For example, allowing a user to select stations from a visual representation of the underground. Such an interface can be developed more easily with an object-oriented language. Therefore, a good solution would be to implement the declarative part of the application in PROLOG and the user interface in JAVA.

3.1.1 Prolog-Side Artefacts

The first stage of the solution consists in expressing our knowledge about an underground system in terms of logic predicates. Most of the code in this section has been taken ‘as is’ from [55].

In the original example, a fragment of the London Underground is represented as a PROLOG logic theory. However, we introduce an interesting variation. Instead of implementing it in plain PROLOG, we use LOGTALK, previously

described in section 2.6. This allows us to profit from a reduced paradigm mismatch when integrating this module with a JAVA-side component.

An underground system is composed by a set of stations connected by means of lines. This is illustrated in figure 3.1.

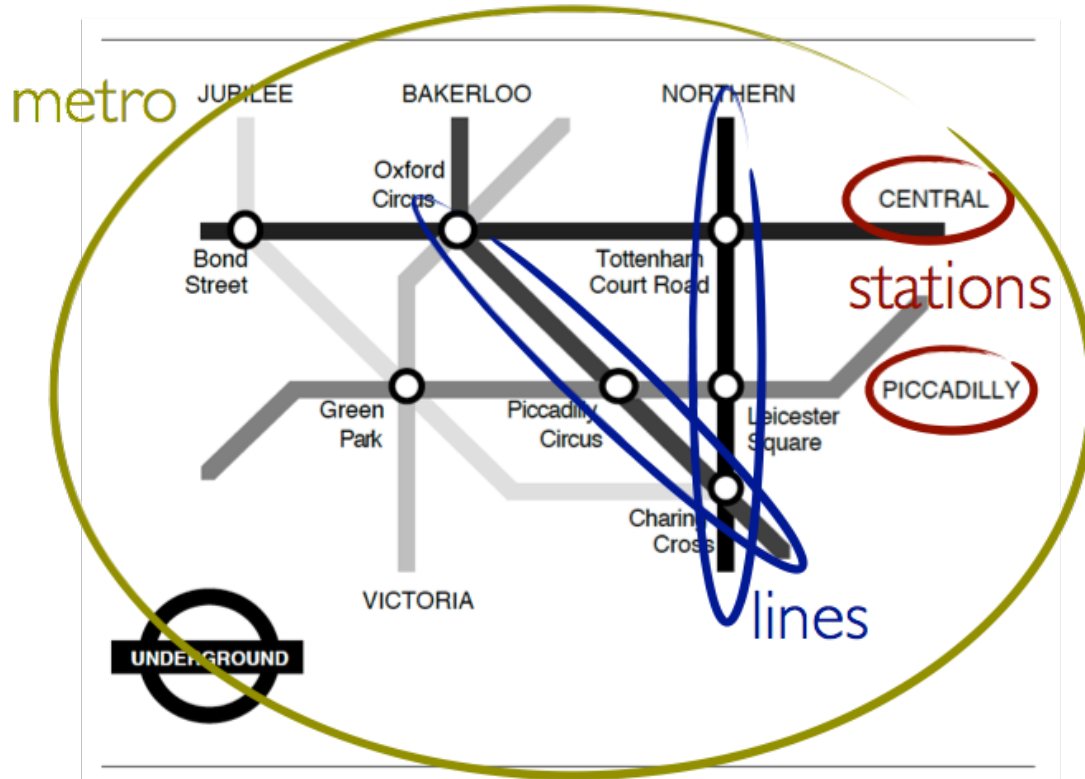


Figure 3.1: The London underground main entities.

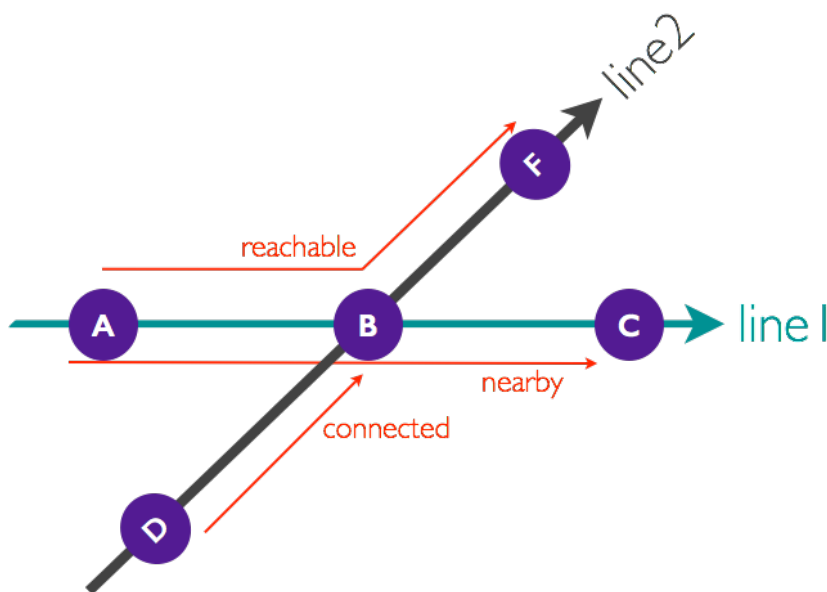


Figure 3.2: Relations between stations.

In our universe of discourse there are three relevant relations between lines

and stations (figure 3.2): (1) Stations are *connected* to other stations through underground lines; (2) A station is *nearby* another one if there is at most one station in between them; Finally, (3) a station *A* is *reachable* from another station *B* if there is a list of stations *L* that forms a path going from *B* to *A*.

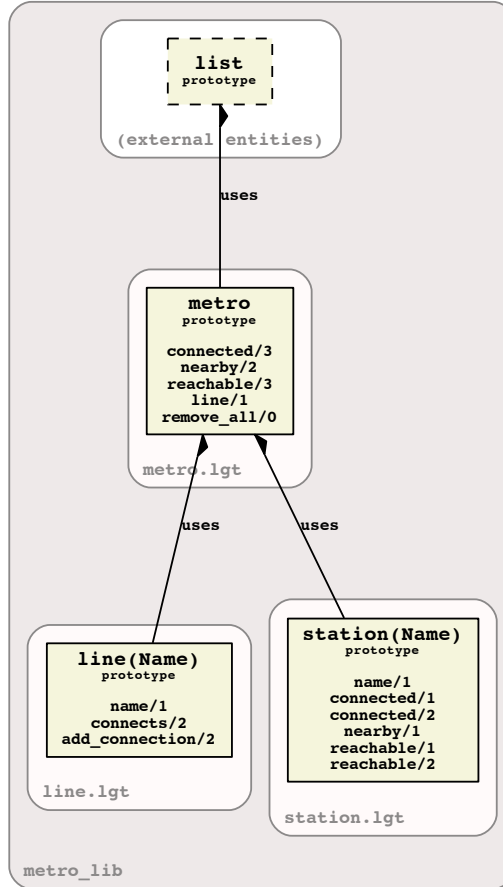


Figure 3.3: Underground example LOGTALK entity diagram.

In the rest of this section we provide an implementation in LOGTALK of the entities illustrated in figures 3.1 and 3.2.

The relationships between these entities and other LOGTALK artefacts is shown in figure 3.3. There are three LOGTALK objects in the `metro_lib` library (*i.e.*, `metro/0`, `line/1` and `station/1`). Each of them is surrounded by a rounded rectangle labelled with the file where those entities are declared (*i.e.*, `metro.lgt`, `line.lgt` and `station.lgt`). The ‘uses’ relation is denoted by a black line ending with a half arrow. Therefore, the figure implies that the objects `line/1` and `station/1` make explicit usage of `metro/0`. The usage of entities external to the library is also depicted. In this case, the `metro/0` object is shown to have a ‘uses’ dependency on `list/0`, which is one of the pre-defined objects defined by LOGTALK.

Code snippet 3.1 shows the LOGTALK definition of the `metro` prototype object. This object encapsulates the knowledge about how stations are connected (lines 8–11), plus the rules for the logic predicates `nearby/2` (lines 16–20) and `reachable/3` (lines 25–29). It also declares a public predicate `line/1` (lines 34–36) answering the name of the registered lines. The implementation of this

predicate makes use of the `list/0` LOGTALK object (line 36), which is pre-defined by LOGTALK in the `types_loader` library. The messages (queries) that the `metro` object can understand are specified using the `public/1` directive (line 3).

```

1 :- object(metro).
2
3 :- public([connected/3, nearby/2, reachable/3, line/1, ...]).
4
5 :- info(connected/3, [
6     comment is 'Station1 is connected to Station2 by means of Line.',
7     argnames is ['Station1', 'Station2', 'Line']]).
8 connected(station(bond_street), station(oxford_circus),line(central)).
9 connected(station(bond_street), station(green_park),line(jubilee)).
10 connected(station(green_park), station(charing_cross),line(jubilee)).
11 ...
12
13 :- info(nearby/2, [
14     comment is 'Station1 is nearby Station2.',
15     argnames is ['Station1', 'Station2']]).
16 nearby(X,Y):-
17     connected(X,Y,L).
18 nearby(X,Y):-
19     connected(X,Z,L),
20     connected(Z,Y,L).
21
22 :- info(reachable/3, [
23     comment is 'Station2 is reachable from Station1 by means of ↔
24         traversing IntermediateStations.',
25     argnames is ['Station1', 'Station2', 'IntermediateStations']]).
26 reachable(X,Y,[]):-
27     connected(X,Y,L).
28 reachable(X,Y,[Z|R]):-
29     connected(X,Z,L),
30     reachable(Z,Y,R).
31
32 :- info(line/1, [
33     comment is 'Name is the name of a Line.',
34     argnames is ['Name']]).
35 line(Name) :-
36     setof(Line, Station1^Station2^connected(Station1,Station2,Line), ↔
37         AllLines),
38     list::member(line(Name), AllLines).
39 ...
40 :- end_object.

```

Snippet 3.1: The `metro` object in LOGTALK.

As explained in section 2.6, messages are sent in LOGTALK using the `::/2` operator. For example, to find out which stations are connected to the *Bond Street* station we can use the query shown in code snippet 3.2.

```

1 metro::connected(station(bond_street), Station, Line).

```

Snippet 3.2: Invoking a LOGTALK method.

LOGTALK and PROLOG both support non-deterministic queries, which allows retrieving all existing solutions for a query using backtracking, and meta-programming, which allows e.g. to construct a list with all solutions to a query. For example, to get a list of all stations connected to *Bond Street* we could write the query shown in code snippet 3.3.

```

1  findall(
2      Station,
3      metro::connected(station(bond_street), Station, Line),
4      Stations
5  ).

```

Snippet 3.3: Invoking a LOGTALK method using the `findall/3` meta-predicate.

Code snippet 3.4 shows the definition of the parametric object `line/1`.

```

1  :- object(line(_Name)).
2
3  :- public([name/1, connects/2, ...]).
4
5  :- info(name/1, [
6      comment is 'Name is the line name.',
7      argnames is ['Name']]).
8  name(Name) :-
9      parameter(1, Name).
10
11 :- info(connects/2, [
12     comment is 'Station1 is connected to Station2 by means of this line.'↵
13     ,
14     argnames is ['Station1', 'Station2']]).
14 connects(Station1, Station2) :-
15     self(Self),
16     metro::connected(Station1, Station2, Self).
17 ...
18 :- end_object.

```

Snippet 3.4: The `line` object in LOGTALK.

The object's sole parameter (line 1) can be retrieved with the method `name/1` (lines 8–9). This object also defines a `connects/2` method (lines 14–16) that answers stations directly connected by the `line` represented by the receiver object. This method implementation is delegated to the `metro` object (line 16).

Our last object is the `station` object (Code snippet 3.5). As for the `line` object, it is also a parametric object having as sole parameter the name of a station (line 1). It defines a method `connected/1` (lines 14–15) that answers if the station is connected with another station (or answers the stations that are connected if the parameter is an unbound variable). `connected/2` (line 20) takes as a second parameter the underground line that connects the stations. The method `nearby/1` (lines 25–27) answers if the station is nearby another station received as a parameter. The method `reachable/1` (lines 32–33) answers if the station received as parameter is reachable from the receiver station object (or answers the reachable stations if the parameter is an unbound variable). The method `reachable/2` (line 38) does the same, but includes in its second argument a list with all the intermediate stations. All these methods are delegated to the `metro` object.

```

1  :- object(station(_Name)).
2
3  :- public([name/1, connected/1, connected/2, nearby/1, reachable/1, ↵
4      reachable/2]).
5
6  :- info(name/1, [
7      comment is 'Name is the station name.',

```

```

7     argnames is ['Name'])).
8     name(Name) :-
9         parameter(1, Name).
10
11 :- info(connected/1, [
12     comment is 'Station is a connected station.',
13     argnames is ['Station'])).
14 connected(Station) :-
15     connected(Station, _).
16
17 :- info(connected/2, [
18     comment is 'Station is a connected station by means of Line.',
19     argnames is ['Station', 'Line'])).
20 connected(Station, Line) :- ...
21
22 :- info(nearby/1, [
23     comment is 'Station is a nearby station.',
24     argnames is ['Station'])).
25 nearby(Station) :-
26     self(Self),
27     metro::nearby(Self, Station).
28
29 :- info(reachable/1, [
30     comment is 'Station is a reachable station.',
31     argnames is ['Station'])).
32 reachable(Station) :-
33     reachable(Station, _).
34
35 :- info(reachable/2, [
36     comment is 'Station is a reachable station by means of traversing ↔
37         IntermediateStations.',
38     argnames is ['Station', 'IntermediateStations'])).
39 reachable(Station, IntermediateStations) :- ...
40 :- end_object.

```

Snippet 3.5: The `station` object in LOGTALK.

Finally, we show the *loader* file of our library. A loader file defines the collection of objects that should be loaded (line 2) when requiring a LOGTALK library.

```

1 :- initialization(
2     logtalk_load([metro, station, line])
3 ).

```

Snippet 3.6: The *load_all.lgt* loader file.

3.1.2 Java-Side Artefacts

In this section we provide an overview of how a JAVA class can be integrated with the logic program described in the previous section. For this, we use JPL, a well known JAVA-PROLOG integration library. We have chosen this library since it is a representative example of the state of the art for integrating an object-oriented language with a non-embedded logic engine. We come back to this example later in chapter 7 to compare it with an alternative mechanism significantly reducing the boilerplate integration code. In chapter 8, we make use of this example again to show an even higher-level integration framework that attempts to make the integration transparent and (semi-)automatic.

We start by illustrating in figure 3.4 the mappings between artefacts in both worlds.

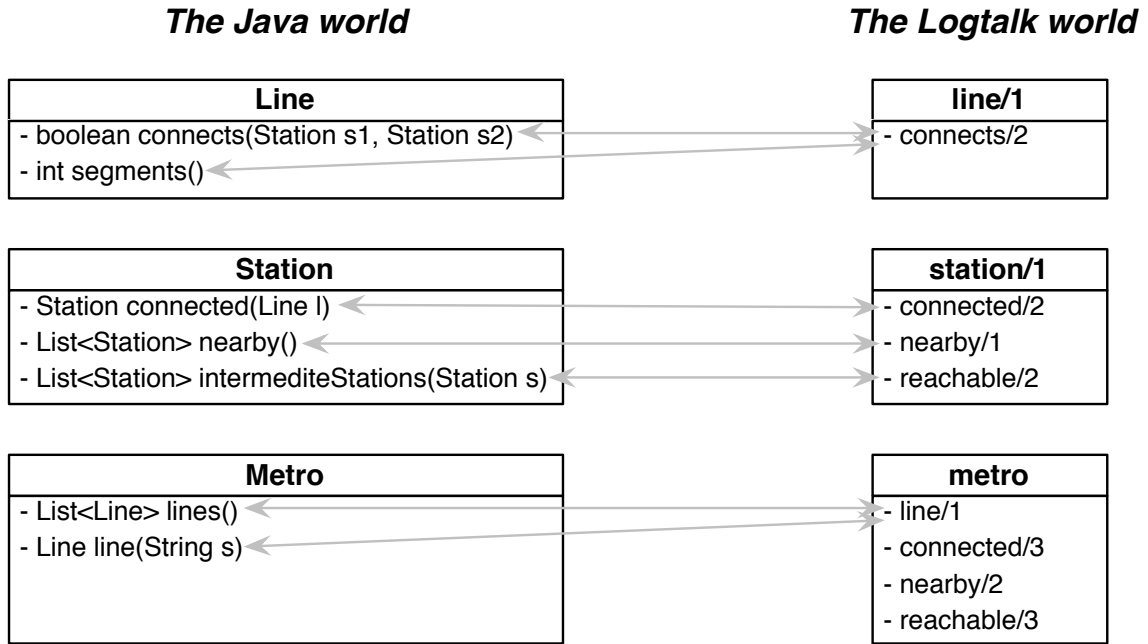


Figure 3.4: Mapping JAVA methods to PROLOG predicates.

Code snippet 3.7 shows an outline of the **Station** JAVA class.

```

1 public class Station {
2     String name;
3     ...
4     // mapping an instance of Station to a logic Term
5     public Term asTerm() {
6         return new Compound("station", new Term[] {new Atom(name)});
7     }
8     // mapping a logic Term to an instance of Station
9     public static Station create(Term stationTerm) {
10         String lineName = ((Compound)stationTerm).arg(1).name();
11         return new Station(lineName);
12     }
13     // mapping a Java method to a Logtalk method
14     public Station connected(Line line) {
15         Station connectedStation = null;
16         String stationVarName = "ConnectedStation";
17         Term[] arguments = new Term[]{new Variable(stationVarName), line←
18             .asTerm()};
19         Term message = new Compound("connected", arguments);
20         Term objectMessage = new Compound("::", new Term[] {asTerm(), ←
21             message});
22         Query query = new Query(objectMessage);
23         Hashtable<String, Term> solution = query.oneSolution();
24         if(solution != null) {
25             Term connectedStationTerm = solution.get(stationVarName);
26             connectedStation = create(connectedStationTerm);
27         }
28         return connectedStation;
29     }
30     // other methods mapped to logic routines
31     ...
32 }

```

Snippet 3.7: A JAVA class interacting with PROLOG by means of JPL.

The class defines how a `Station` instance should be represented on the PROLOG-side by means of the instance method `asTerm()` (lines 5–7). In our example, a station is represented in PROLOG as a compound term using the functor `station` and an atom as single argument representing the name of the station (line 6).

The static method `create(Term)` (lines 9–12) does the opposite: it defines how a station term defined on the PROLOG-side has to be represented as a JAVA object. It takes the first argument of the term representing the station's name (line 10) and uses it to instantiate the `station` class (line 11). In order to keep it simple we ignore error handling verifying that the term has indeed the right structure.

The method `connected(Line)` (lines 14–27) returns a station connected to the receiver by means of the line received as a parameter. First a term representing a LOGTALK message is built on line 18. The arguments of this message are an unbound variable and the term representation of the line object received as parameter (line 17). Then an object message is built on line 19. LOGTALK uses the `::/2` infix operator for sending a message to an object. Its left operand is the receiver (in this case the result of invoking the `asTerm()` method) and the right operand is a LOGTALK message with its arguments. Thus, the query created on line 20 is interpreted in LOGTALK as the message:

```
station(station_name)::connected(ConnectedStation, line(line_name)).
```

Once the query has been constructed, we request its first solution on line 21. The binding for the variable sent as first argument to the LOGTALK method is collected from the solution on line 23. This variable has been bound to a PROLOG term representing a station. On line 24 we create the JAVA representation of this station object and we return it on line 26.

3.1.3 Discussion

As illustrated by this example, for each query, the programmer must write the necessary code to convert multiple JAVA objects to a convenient PROLOG term representation (*e.g.*, lines 17 and 19) and to convert PROLOG terms back to JAVA objects (*e.g.*, line 24).

Larger applications often require writing a large number of (possibly context-dependent) conversion routines. This is an error-prone activity that often produces ad-hoc conversion code tangled with other aspects of the code.

In chapter 8 we introduce a framework that overcome these problems. This framework has as objective to abstract the programmer from the integration concern. In this way, the 'glue code' that communicates the two worlds is separated from the main concern of the application and, in certain cases, generated behind the curtains.

In chapter 9 we discuss an improved implementation by means of this framework of the example presented in this section.

3.2 An Application for Querying Geographical Data

While the previous example was focused on the integration from the object-oriented perspective, the example introduced in this section is centred on the other side.

This case study discusses the MAPQUERY [24] application. MAPQUERY allows to query and visualise geographical data by means of PROLOG.

A pre-condition is that geographical data adhering to the Open Street Map (OSM) [69] format in XML [85] needs to be imported first by the application. Code snippet 3.8 shows a small fragment of a dump of data corresponding to the city of Brussels.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <osm version="0.6" generator="CGImap 0.0.2">
3   <bounds minlat="50.8319000" minlon="4.3355000" maxlat="50.8599000" ↵
      maxlon="4.3708000"/>
4   <node id="145324" lat="50.8468554" lon="4.3624415" user="BenoitL" uid=↵
      ="101145" visible="true" version="18" changeset="7864502" ↵
      timestamp="2011-04-14T20:01:32Z">
5     <tag k="highway" v="traffic_signals"/>
6   </node>
7   <node id="145325" lat="50.8454258" lon="4.3617097" user="moyogo" uid=↵
      "246" visible="false" version="16" changeset="672609" timestamp=↵
      "2009-02-26T13:54:09Z"/>
8   <node id="145332" lat="50.8424618" lon="4.3592579" user="moyogo" uid=↵
      "246" visible="true" version="6" changeset="627307" timestamp="↵
      2009-02-23T06:47:09Z">
9     <tag k="highway" v="traffic_signals"/>
10  </node>
11  ...
12  <way id="99502847" user="BenoitL" uid="101145" visible="true" version=↵
      ="1" changeset="7275543" timestamp="2011-02-13T13:53:12Z">
13    <nd ref="192586557"/>
14    <nd ref="192586470"/>
15    <nd ref="1150934232"/>
16    ...
17    <tag k="highway" v="footway"/>
18  </way>
19  <way id="127346641" user="toSc" uid="246723" visible="true" version="↵
      1" changeset="9103520" timestamp="2011-08-23T13:06:47Z">
20    ...
21  </way>
22  ...
23 </osm>

```

Snippet 3.8: OSM raw data.

This data includes *nodes*, which are points of interests having a unique identifier and with a precise pair of *coordinates* (latitude and longitude). Some of those nodes may include *tags* that denote their type and properties. For example, in code snippet 3.8 two nodes are identified as traffic signals by their tags (lines 5 and 9).

Other artefacts described by this document are *ways*. A way is a collection of nodes and, as such, have a unique identifier and may include tags that identify their nature. For example, code snippet 3.8 identifies a way with id *99502847* and that is identified as a *footway* (line 17).

We will not enter into many details of how an OSM document can be parsed

and translated into a logic theory since this has been left outside the scope of this dissertation. Intuitively, the XML document is parsed using the Simple API for XML (SAX) library [87] to generate events for each new OSM element read. Afterwards, a listener of these events translates them to events describing predicates which are asserted into the PROLOG engine.

We describe the main artefacts on the PROLOG and JAVA-side in the rest of this section.

3.2.1 Java-Side Artefacts

In this section we overview the artefacts on the JAVA-side that should be integrated within PROLOG. MAPQUERY provides a visual JAVA interface developed using the JAVAFX library [137]. As shown in figure 3.5, this interface embeds a map that can be controlled to show nodes and ways.

Figure 3.6 shows the main classes of MAPQUERY on the JAVA-side. At the top of the figure there is a **Taggable** interface implemented by both the **Node** and **Way** classes. This interface declares a method returning a map of tag names to tag values. As shown in the figure, a **Way** is a composition of **Nodes** and a **Node** has exactly one pair of coordinates.

The class diagram also introduces the **MapBrowser** class. This class models the map component shown in figure 3.6 and includes the nodes and ways that are drawn on it.

The loading of OSM data into a PROLOG engine is accomplished by an utility class **OsmDataLoader** not shown in the previous class diagram. This class provides a routine for loading the geographical data from a file adhering to the OSM format into a PROLOG engine, as shown in listing 3.9 (lines 9–11).

```

1 public class OsmDataLoader {
2     ...
3     private PrologEngine prologEngine;
4
5     public OsmDataLoader(PrologEngine prologEngine) {
6         this.prologEngine = prologEngine;
7     }
8
9     public void load(File osmFile) {
10         //loads content of osmFile into prologEngine
11     }
12     ...
13 }

```

Snippet 3.9: An utility class for loading OSM data into a PROLOG engine.

3.2.2 Prolog-Side Artefacts

As in the previous example, we have modelled our artefacts on the logic side using LOGTALK. Figure 3.7 shows a simplified entity diagram illustrating the main entities and their relations. The only new element in comparison with the diagram shown in figure 3.3 is the existence of the LOGTALK category **taggable**

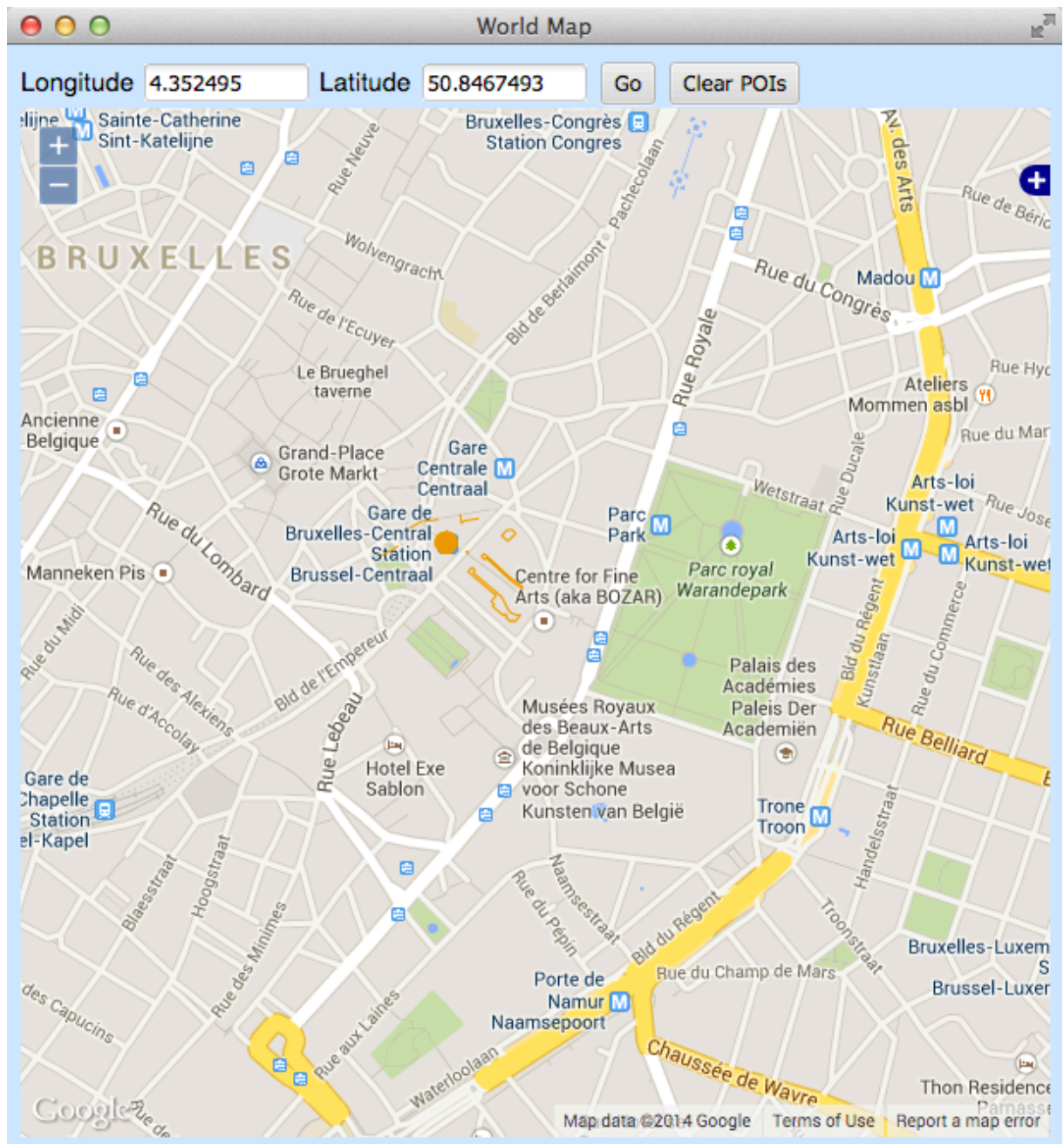


Figure 3.5: The MAPQUERY interface.

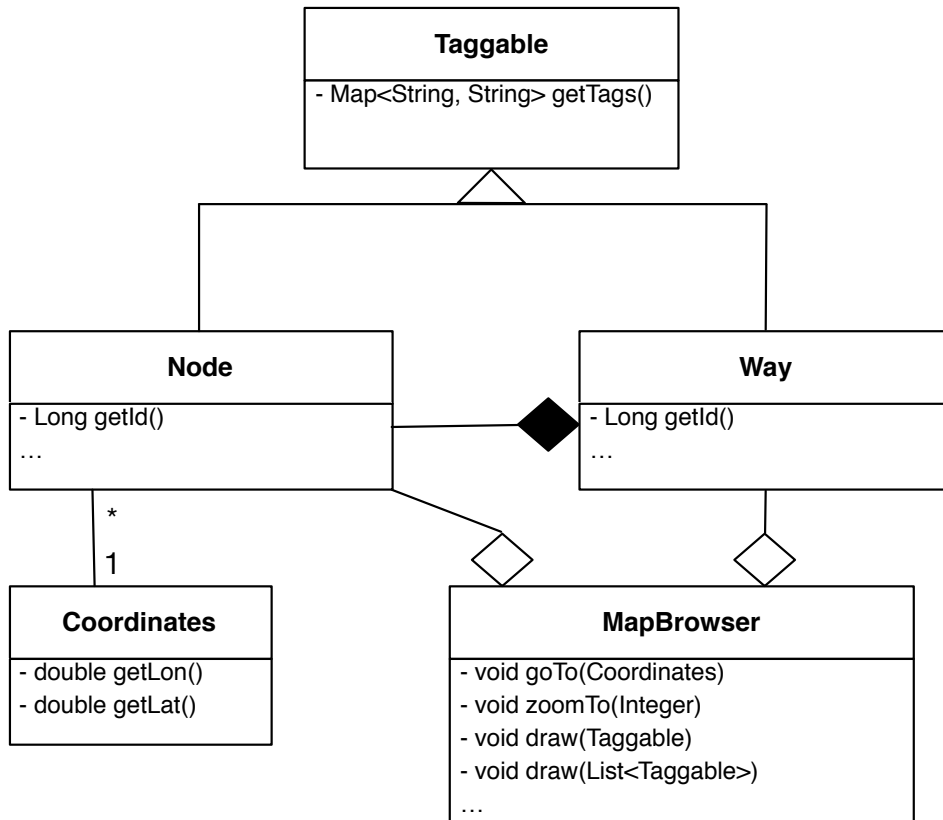


Figure 3.6: The MAPQUERY class diagram.

/1. The black lines with a square at the end denote that such category is imported by the objects way/3 and node/3.

The `coordinates/2` object, illustrated in code snippet 3.10, is parameterised with a longitude and latitude (line 1) and, amongst others, provides methods for determining if a pair of coordinates is nearby other coordinates according to some defined distance (lines 20–22), for calculating the distance between two pairs of coordinates (line 27), and so on.

```

1 :- object(coordinates(_Lon,_Lat)).
2
3 :- public([lon/1, lat/1, distancekm/2, distancem/2, near/2]).
4
5 :- info(lon/1, [
6     comment is 'Lon is the longitude of this pair of coordinates.',
7     argnames is ['Lon']]).
8 lon(Lon) :-
9     parameter(1, Lon).
10
11 :- info(lat/1, [
12     comment is 'Lat is the latitude of this pair of coordinates.',
13     argnames is ['Lat']]).
14 lat(Lat) :-
15     parameter(2, Lat).
16
17 :- info(near/2, [
18     comment is 'This pair of coordinates is near to coordinates ←
19         Coordinates according to the distance in kilometers Km.',
20     argnames is ['Coordinates', 'Km']]).
21 near(Coordinates, Km) :-
22     distancekm(Coordinates, D),
  
```

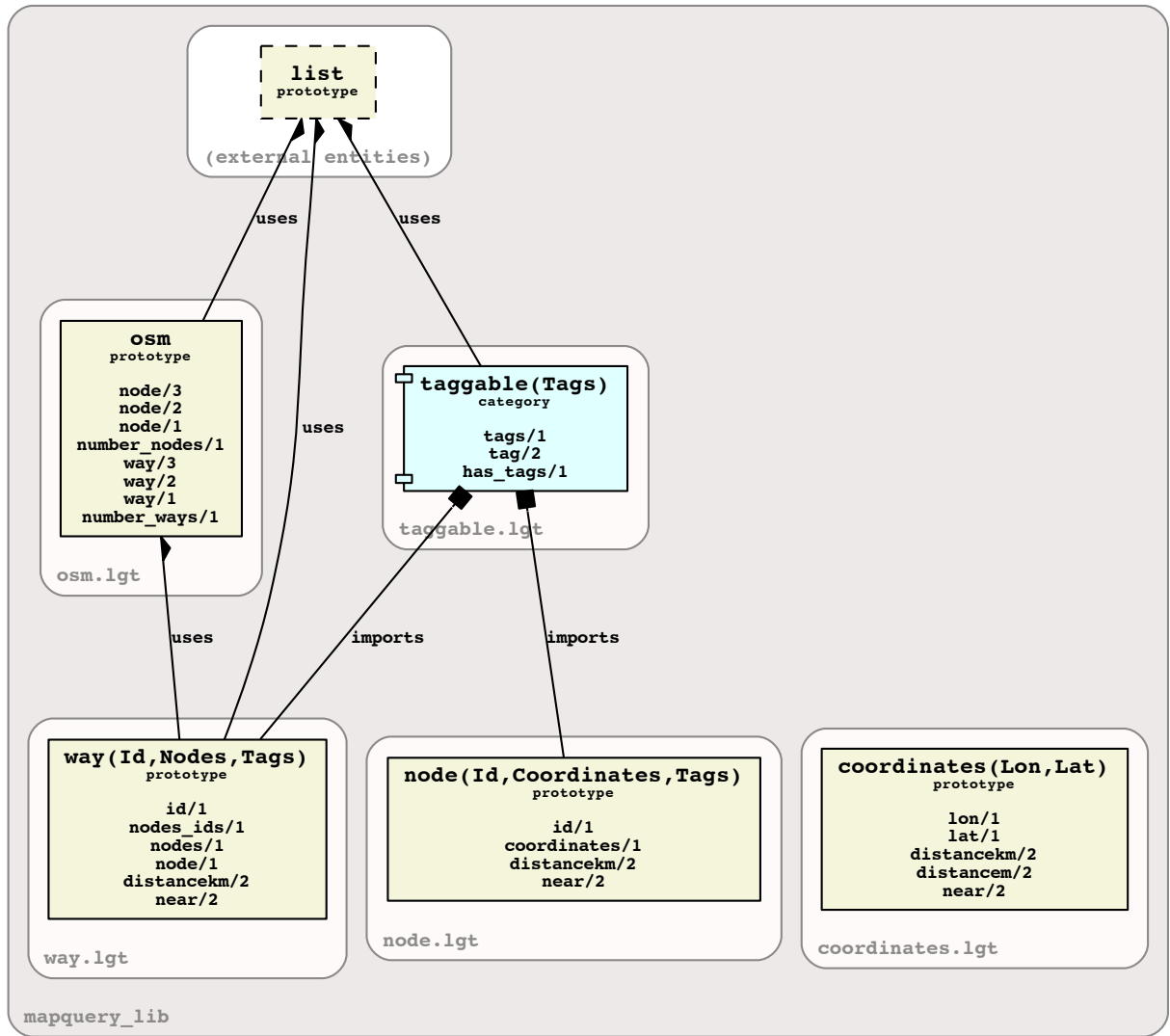


Figure 3.7: Simplified MAPQUERY LOGTALK entity diagram.

```

22     D=<Km.
23
24     :- info(distancekm/2, [
25         comment is 'D is the distance in kilometers to coordinates ←
26             Coordinates.',
27         argnames is ['Coordinates', 'D']]).
28     distancekm(Coordinates, D) :- ...
29     ...
30 :- end_object.

```

Snippet 3.10: The coordinate/2 LOGTALK object.

The `taggable/1` category allows to encapsulate in one entity predicates related to the management of tags. The parameter of this category is a list of tags, where each element has the form `Key-Value`, indicating the name and value of an arbitrary tag.

This category provides predicates for querying if a `Key-Value` pair is part of the tags (lines 14–16) or if a list of tags is part of them (lines 21–23).

```

1 :- category(taggable(_Tags)).
2
3 :- public([tags/1, tag/2, has_tags/1]).

```

```

4
5 :- info(tags/1, [
6     comment is 'Tags is the tags in this taggable.',
7     argnames is ['Tags']]).
8 tags(Tags) :-
9     parameter(1, Tags).
10
11 :- info(tag/2, [
12     comment is 'Value is a tag value associated to the tag name Key.',
13     argnames is ['Key','Value']]).
14 tag(Key, Value) :-
15     tags(Tags),
16     list::member(Key-Value,Tags).
17
18 :- info(hastags/1, [
19     comment is 'SomeTags is a list of tags owned by this taggable.',
20     argnames is ['SomeTags']]).
21 has_tags(SomeTags) :-
22     forall(list::member(Key-Value, SomeTags),
23         tag(Key, Value)).
24
25 :- end_category.

```

Snippet 3.11: The taggable/1 LOGTALK category.

Code snippet 3.12 shows a partial implementation of the `node/3` object. This object imports the `taggable/1` category and is parameterised with the node Id, its coordinates and tags (line 1). It provides methods similar to the `coordinates/2` object to determine the node's geographical position with respect to certain coordinates.

```

1 :- object(node(_Id,_Coordinates,_Tags), imports(taggable(_Tags))).
2
3 :- public([id/1, coordinates/1, distancekm/2, near/2]).
4
5 :- info(id/1, [
6     comment is 'Id is the identifier of this node.',
7     argnames is ['Id']]).
8 id(Id) :-
9     parameter(1, Id).
10
11 :- info(coordinates/1, [
12     comment is 'Coordinates is the pair of coordinates of this node.',
13     argnames is ['Coordinates']]).
14 coordinates(Coordinates) :-
15     parameter(2, Coordinates).
16
17 :- info(near/2, [
18     comment is 'This node is near to coordinates Coordinates according to↵
19         the distance in kilometers Km.',
19     argnames is ['Coordinates', 'Km']]).
20 near(ThatCoordinates, Km) :- ...
21
22 :- info(distancekm/2, [
23     comment is 'Km is the distance in kilometers to coordinates ↵
24         Coordinates.',
24     argnames is ['Coordinates', 'Km']]).
25 distancekm(ThatCoordinates, Km) :- ...
26
27 :- end_object.

```

Snippet 3.12: The node/3 LOGTALK object.

Code snippet 3.14 shows a partial implementation of the `way/3` object. Like

the `node/3` object, it imports the `taggable/1` category. It is parameterised with the way Id, the list of nodes (the node Ids in fact) that compose the way and the way's tags (line 1).

A `way/3` object exposes the nodes that compose it through the public predicate `node/1` (line 21). As previous objects, it provides predicates to determine its geographical position with respect to certain coordinates.

```

1 :- object(way(_Id,_NodesIds,_Tags), imports(taggable(_Tags))).
2
3 :- public([id/1, nodes_ids/1, nodes/1, node/1, distancekm/2, near/2]).
4
5 :- info(id/1, [
6     comment is 'Id is the identifier of this way.',
7     argnames is ['Id']
8 ]).
9 id(Id) :-
10     parameter(1, Id).
11
12 :- info(nodes_ids/1, [
13     comment is 'NodesIds is a list with the identifiers of the nodes ↵
14         composing this way.',
15     argnames is ['NodesIds']).
16 nodes_ids(NodesIds) :-
17     parameter(2, NodesIds).
18
19 :- info(node/1, [
20     comment is 'Node is a node composing this way.',
21     argnames is ['Node']]).
22 node(Node) :- ...
23
24 :- info(near/2, [
25     comment is 'This way is near to coordinates Coordinates according to ↵
26         the distance in kilometers Km.',
27     argnames is ['Coordinates', 'Km']]).
28 near(ThatCoordinates, Km) :- ...
29
30 :- end_object.

```

Snippet 3.13: The `way/3` LOGTALK object.

Finally, the `osm/0` object acts as a centralised database containing information about existing nodes and ways. Note that the `way/3` and `node/3` predicates are declared as dynamic (line 4), so that new ways and nodes can be asserted after the `osm/0` object has been loaded.

```

1 :- object(osm).
2
3 :- public([node/3, node/2, node/1, number_nodes/1, way/3, way/2, way/1, ↵
4     number_ways/1]).
5 :- dynamic([way/3, node/3]).
6
7 :- info(node/1, [
8     comment is 'Node is a node.',
9     argnames is ['Node']]).
10 node(Node) :- ...
11
12 :- info(way/1, [
13     comment is 'Way is a way.',
14     argnames is ['Way']]).
15 way(Way) :- ...
16 :- end_object.

```

Snippet 3.14: The way/3 LOGTALK object.

Querying the OSM Data

Code snippet 3.15 shows a query obtaining all the ways (*e.g.*, streets) near the ‘*Bruxelles-Central*’ railway station in the city of Brussels. On line 1 we obtain a node registered in the OSM database. We verify that it has the tag *railway-station* (line 2) and that its name in french is ‘*Bruxelles-Central*’ (line 3). We ask for the coordinates of this node on line 4. Then we ask for a way (line 5) with the condition that it is near to the node we found, with a nearness criterion of 100 meters (line 6).

```
1 osm::node(Node),
2 Node::has_tags([railway-station]),
3 Node::has_tags(['name:fr'-'Bruxelles-Central']),
4 Node::coordinates(Coordinates),
5 osm::way(Way),
6 Way::near(Coordinates, 0.1)
```

Snippet 3.15: Querying OSM data.

Although not being a very precise query (in the OSM format any collection of nodes can be a way, even, for example, a collection of buildings), it illustrates how the main LOGTALK entities can interact to find places with particular attributes.

From Prolog to Java

If a programmer would like to visualise in the JAVA interface (figure 3.5) the nodes and ways that were queried in code snippet 3.15, a JAVA routine should be invoked from within PROLOG. An example of this, using the JPL library, is illustrated in code snippet 3.16.

The `jpl_call/4` predicate receives four arguments: (1) the receiver of a method call; (2) a method name; (3) a list of arguments; and (4) the method return value. We assume that the first argument, the `Map` variable, is a reference to the JAVA map interface. The method name indicated in the second argument is `draw`. In the method arguments list, the `Way` variable is between curly braces to instruct JPL to pass it by using its term representation (an instance of `jpl.Term` on the JAVA-side). Finally, the return value of the method is ignored by putting an anonymous variable as fourth argument of the `jpl_call/4` predicate.

```
1 jpl_call(Map, draw, [{Way}], _).
```

Snippet 3.16: Visualising OSM data.

Discussion

The previous example illustrated two problems. First, a programmer should refer to a predicate (*i.e.*, `jpl_call/4`) explicitly bridging the two worlds. This implies that the integration concern (*e.g.*, the call to a foreign routine) is not hidden but tangled with the real problem the programmer is trying to solve (*e.g.*, show one or more artefacts on a map). D’hondt observes that this tangling hinders the replacement of a foreign call with a native implementation, or vice versa [46].

Although the foreign call could be wrapped in another predicate so the foreign call is not explicit, this requires the programmer to write boilerplate code (*e.g.*, the wrappers).

A second problem concerns the transformation of entities from their term representation (*e.g.*, a compound having as functor `way/1`) to an object (*e.g.*, an instance of the `Way` class). With a library such as JPL this transformation needs to be explicitly invoked in the receiver of the foreign call on the object-oriented side.

We claim that this transformation should be transparent to the programmer. In this way, a programmer in the object-oriented world should not have to write programs in terms of explicit concepts (*e.g.*, term types and queries) from the logic language. From the logic language perspective, a logic programmer should not be required to write programs explicitly modelling concepts (*e.g.*, object state and method calls) from the object-oriented language. Our approach for reaching such transparency is discussed in chapter 8 and concrete examples are provided in chapter 9.

3.3 Chapter Summary

This chapter has illustrated two case studies that will be referred to during the rest of this dissertation. The first case study focusses on the integration from the object-oriented perspective, while the second covers the logic programming perspective.

For each case study we provided small examples illustrating how the integration could be accomplished using a well-known integration library. In addition, we discussed the limitations of that technique.

One of the main limitations concerns the non-transparent conversions between inter-language artefacts (*i.e.*, logic terms to objects and back). In the next chapter we present a conceptual analysis of this and other integration problems. The concrete design and implementation of a library which tackles these problems is described in chapter 6 and 7.

4 A Conceptual Model for Bidirectional Integration between a Logic and a Statically-Typed Object-Oriented Language

Science is a way of thinking much more
than it is a body of knowledge.
—Carl Sagan

The related work introduced in chapter 2 compared techniques providing different levels of abstractions for integrating hybrid logic and object-oriented programs. That chapter also illustrated a trade-off between integration techniques relying on embedded and non-embedded logic engines.

The former techniques are often closer to a full oblivious integration thanks to the two program execution environments sharing the same memory space. This reduces the complexity of several integration problems (*e.g.*, dealing with object state).

The latter techniques, on the other hand, are more common in industry, because they have access to a significant body of well-tested logic libraries and are often more performant than their embedded counterparts.

In this chapter, before diving into the architecture and the implementation of our proposed solution, we introduce a conceptual model for the *portable* integration (*i.e.*, compatible with both embedded and non-embedded logic engines) between a statically-typed object-oriented language and a logic language. In this model, we assume the object-oriented language to be stateful, which implies that object properties can change over time and that the integration approach needs to address the critical problem of handling state. Our model also takes into account aspects regarding the behavioural integration of these languages. Purposefully, the discussion in this chapter is kept at an abstract level. The architecture of our approach follows in chapter 5. The description of a concrete implementation of this architecture is given in chapters 6, 7 and 8.

4.1 Design Trade-Offs

Designing an oblivious integration approach portable to distinct architectures (*e.g.*, a logic engine embedded or not in the object-oriented environment) is far from trivial. This is because such an approach should not make any of the simplifying assumptions typically made by existing techniques to reach an oblivious integration (*i.e.*, artefacts sharing the same memory space).

Possible design strategies for an approach supporting both oblivious integration and compatible with non-embedded logic engines thus need to strike a delicate balance.

Bottom-up design strategies (*i.e.*, starting from the low-level integration problems towards the final API) risk to end up with a library not improving on existing integration techniques. Since the library constraints are influencing the design decisions from the beginning this may influence the entire way in which the problem is approached.

On the other hand, following a top-down approach (*i.e.*, starting from an ‘ideal’ oblivious integration design and subsequently attempting to refine it until it incorporates all low-level constraints) risks to end up with a solution that may not be implementable in real business scenarios.

Given that our objective is not to implement yet another integration library with similar limitations as the existing ones but rather to provide a *portable* library supporting oblivious integration, we followed the second alternative of using a top-down approach. Our objective is to sacrifice, in our quest for portability, as little advanced integration features as possible.

4.2 Defining a Common Model

Brichau et al. [9, 66] observed that the difficulty of integrating object-oriented and logic programs is caused by the different concepts that each paradigm provides. Mapping concepts between these paradigms is far from trivial, given that several artefacts or concepts that exist in one paradigm do not exist in the other. This has been referred to as a *paradigm mismatch*.

D’Hondt [46] proposes to solve this problem by writing integration code in a hybrid language amalgating features of the two languages being integrated. Since the hybrid language used for the integration incorporates notions of both integrated languages, the paradigm mismatch is reduced. The integration code written in this hybrid language is referred to in that approach as a *paradigmatic buffer*.

In this dissertation we follow a different approach. Instead of using a third multi-paradigm language for accomplishing the integration, we attempt to reduce the paradigm mismatch by approaching, as much as possible, the two paradigms. This is possible because the object-oriented paradigm can be considered as orthogonal to most existing programming paradigms such as functional programming (*e.g.*, CLOS [122]), logic programming (*e.g.*, LOGTALK) or imperative programming (*e.g.*, SMALLTALK or JAVA) [104].

Therefore, the object-oriented programming paradigm can act as a unified model and provide a common vocabulary and abstractions for the integration of programs.

4.3 Capturing Object State: The Critical Integration Point

Dealing with mutable object state (*i.e.*, object dynamics [1]) has been recognised in the past as the most critical point when integrating objects with logic [1, 104]. This is because the notion of object mutability conflicts with the declarative nature of a logic language. Most other aspects of object-oriented programming (*e.g.*, encapsulation, polymorphism, aggregation, inheritance, classes or prototypes) can be added to a logic programming language with relative ease (*e.g.*, as done by LOGTALK).

For integration approaches where the logic and the object-oriented program share the same memory space, this problem can be solved trivially. For example, the logic language can be extended to support a new kind of logic term encapsulating an object from the object-oriented world (*e.g.*, SOUL, JINNI, LEANPROLOG). Since this new term encapsulates (or is, depending on the implementation) a reference to an object on the object-oriented side, there is no need for a special mechanism to synchronise the object state with its logic term reification. Any change to the object state will be automatically reflected in the logic world, since there is a *causal connection* between the actual object and its term reification. An additional advantage of maintaining causality is that queried objects do not have to be reconstructed based on their term reification, since the terms reifying them are just wrappers of the objects themselves.

A disadvantage of this approach, however, is that it is not portable. It requires that the two worlds share the same memory space and the addition of new data types in the logic language.

On the other hand, representing object state in logic engines non-embedded in an object-oriented environment is far from trivial. Different aspects should be considered such as maintaining object identity, the opacity of the representation on the logic side and the objects' life span.

In the remainder of this section we identify and discuss the different dimensions to be considered regarding how to represent object state in the logic language (*fig.* 4.1). These dimensions have been extracted and generalised from existing solutions to this problem in logic languages (*e.g.*, PROLOG and SOUL) and other inter-language representation domains (*e.g.*, Google's GSON library [61]).

4.3.1 Object Reification

A first important dimension is how objects can be represented on the logic side. Several integration libraries allow to reify objects in a logic language using a

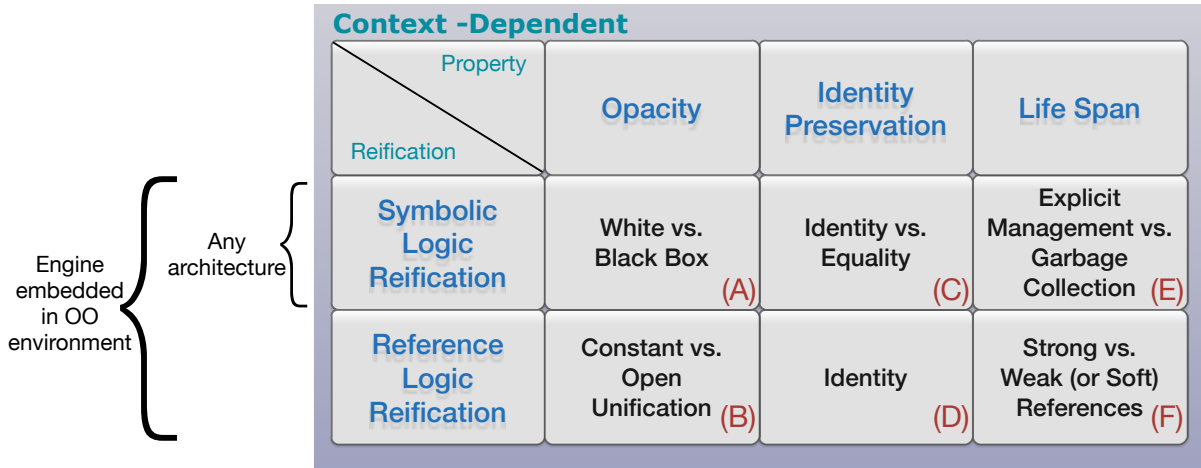


Figure 4.1: Reference Management Dimensions.

symbolic term representation (*e.g.*, JPL [119], INTERPROLOG [14], JASPER [17] or the CIAO Java interface [11]). As shown in figure 4.1, such approach has the advantage of not relying on any specific logic engine architecture.

An example of a symbolic reification was shown in code snippet 3.7 using the JPL library. In this example, a term reifying an instance of a class `Station` was a compound with a single argument of the form `station(station_name)`.

Definition 4.1. A *Symbolic Logic Reification* of an object is accomplished by reifying such object using the existing types in the logic language. This reification strategy is portable across different architectures.

Alternatively, embedded logic engines may support the direct storage of object references (*i.e.*, the object itself, not a symbolic representation of it) (*e.g.*, JINNI, LEANPROLOG and SOUL). An advantage of this reification scheme is that there are no performance penalties associated to the marshalling/unmarshalling of objects to/from the logic engine.

An example of a reference reification was shown in code snippet 2.25 using the SOUL language. In that example, a logic predicate could either unify a logic variable to an object reference (lines 1–3) or receive an object reference as an argument (lines 5–7).

Definition 4.2. A *Reference Logic Reification* of an object is accomplished by reifying such object using a special logic type wrapping the object. This reification strategy is only enabled if the object-oriented and logic environments share the same memory space.

4.3.2 Opacity of the Representation

A second important dimension is the degree of opacity of the representation (*i.e.*, the degree of data exposed) when reifying an object reference in the logic language. For symbolic term representations (*fig.* 4.1 A), frequently a

fine-grained reification of the internal object structure (*i.e.*, a white box representation) is desired. For example, JTRANSFORMER [80] allows to reason over the structure of terms reifying objects modelling a JAVA Abstract Syntax Tree (AST). Therefore, terms reifying an AST should expose part of their structure.

As another example, in the previous section code snippet 3.7 made use of symbolic reification for representing instances of the `Station` class in the logic side. This reification is also a white box representation, since the object data (*i.e.*, the name of the station) is exposed in such representation.

Definition 4.3. *A **White Box Logic Reification** exposes the structure of the object in its term representation. Therefore, it allows to make use of logic programming mechanisms (e.g., unification and pattern matching) for reasoning over the structure of reified objects.*

Alternatively, if inspecting the object's structure on the logic side is not required, having an opaque reference (*i.e.*, a black box representation) to the corresponding object is preferable (*e.g.*, an opaque reference to a GUI component on the object-oriented side). In those cases, an automatic mechanism to generate opaque term representations of objects is desirable.

As an example, the JPL library discussed in our case study in section 3.1.2 allows to express a reference to a JAVA object by means of the compound term `jref/1`. The first argument of this term is an opaque identifier of the object reference. Therefore, this representation of an object does not expose its state on the logic side, being considered a black box logic reification.

Definition 4.4. *A **Black Box Logic Reification** does not expose the internals of an object in its term representation. Special predicates should be used to query object properties on the logic side.*

When the logic engine is embedded in a object-oriented environment, a more direct kind of reference to objects can be established (*fig. 4.1 B*). In the simplest case, the object reference can be considered and unified as a special constant term.

Definition 4.5. *A **Constant Unification of Object References** implies that objects are considered as constants and unify as such on the logic side.*

An example of this can be seen in JINNI. Code snippet 2.24 showed an example of how the `invoke_java_method/3` predicate can be used to obtain the result of a JAVA method invocation in PROLOG. Such result can be an arbitrary JAVA object. When unifying two JAVA objects in JINNI, the `equal` relation is used, basically as if they were PROLOG constants.

But we may want to combine the best of both worlds and have direct references to the actual objects, while still allowing logic programs to reason over the internal structure of such objects. Approaches such as SOUL [114] have achieved this through the mechanism of *open unification* [10]. This approach consists in allowing the programmer to customise not the term representation

of an object, but rather its unification mechanism. In a nutshell, the unification mechanism is opened up so that objects are not regarded as constants but can be unified with structured logic terms of the right form.

As an example, in the SOUL language such an open unification is accomplished by allowing the programmer to override a unification method in any SMALLTALK class, which determines how the unification should be accomplished in instances of those classes.

Definition 4.6. *An **Open Unification of Object References** implies that a programmer can define, per object, how objects unify with other objects or logic terms on the logic side.*

4.3.3 Object Identity Preservation

For non-embedded engines, a programmer needs to decide if an object reification as a logic term should preserve its identity when the term is translated back to an object (fig. 4.1 C).

In certain cases, keeping track of the original reference is required to guarantee the expected behaviour of the program (e.g., if the reference points to a GUI component). Furthermore, passing around symbolic representations of object references is often more efficient than marshalling and unmarshalling large objects.

Definition 4.7. *An **Identity Preserving Reification** allows to preserve an object reference when its term reification is translated back to the object-oriented environment.*

In other scenarios, however, it is not important to preserve an object identity (e.g., usually it is not relevant to preserve the identity of strings) and a different reference, considered equal or equivalent to the original object, is an acceptable outcome. An example of this can be observed in code snippet 3.7. There we illustrated how instances of a `Station` class are reified as the compound term `station/1` (lines 5–7). However, when this compound is translated back to an object (lines 9–12), a new instance of `Station` is created (line 11). Hence the identity of the original `Station` instance is not preserved.

Definition 4.8. *An **Equality Preserving Reification** does not maintain the identity of an object when it is translated back to the object-oriented environment. Instead, the translated object is equivalent or equal to the original according to some notion of equality.*

Note that the need for preserving the original object identity is orthogonal to the required opacity of the representation. I.e., independently of whether the reference should be preserved or not, the programmer should still be able to decide on the best representation of the object on the logic side.

For logic engines embedded in the object-oriented environment (fig. 4.1 D) object references are preserved automatically since the term wraps the object ‘as is’.

4.3.4 Reference Life Span

A fourth dimension is the life span over which the logic term reification of an object reference remains valid. We define this life span as the interval during which a term reification can be converted back to an object.

Life Span of a Symbolic Reification

For a symbolic term representation, the life span of a symbolic reification can be either explicitly delimited or rely on the garbage-collection mechanisms existing in one of the languages to integrate (*fig. 4.1 E*).

For an explicitly delimited life span, a symbolic term representation of an object will remain valid until explicitly discarded, even if it is not explicitly referenced in the object-oriented program (*i.e.*, normally to be scheduled for garbage collection). As an example, an association between an object and its term representation can be discarded by means of the invocation of a routine allowing to ‘forget’ such relation.

Definition 4.9. *An **Explicit Reference Life Span** management allows a programmer to explicitly delimit the life span over which the logic reification of an object reference can be converted back to the original reference.*

Alternatively, if the reference life span is automatically delimited by the garbage-collection mechanisms existing in one of the languages to integrate, the programmer does not have to worry about explicitly maintaining a mapping between terms and references. As an example, the reification of a reference to an application window should remain valid as long as the window is open. Therefore, nothing should prevent the window resources to be collected once it is closed. In this case, the reification of the reference on the logic side will be valid only while the object has not been disposed (*i.e.*, the window is not closed).

A similar behaviour can be implemented relying on the garbage collector mechanism of the logic side, if available. For example, certain logic engines (*e.g.*, SWI) allow a programmer to add hooks to the native atom garbage collection mechanism. In this way, an object can be associated with a term containing an atom serving as object identifier. Then, a program can be notified when the atom is collected on the logic side. At that moment, the reference can be disposed if no further references to it exist.

Definition 4.10. *A **Garbage Collected Reference Life Span** management allows a programmer to rely on the existing garbage collection mechanisms for determining the life span over which the logic reification of an object reference can be converted back to the original reference.*

Life Span of a Wrapped Reference Reification

For a reification wrapping an object reference (*fig. 4.1 F*), the programmer may require to keep alive the reference from the object-oriented world as long as it

is referenced on the logic side (*i.e.*, a strong reference).

Definition 4.11. *A **Strong Reification of a Reference** wraps an object reference in such a way that it prevents the reference from being garbage-collected.*

However, in certain scenarios an object referenced by a logic theory should not prevent it from being garbage collected (*e.g.*, the reference points to a disposed GUI component). In that case, the reference should be invalidated when it is reclaimed by the garbage collector.

Definition 4.12. *A **Weak Reification of a Reference** wraps an object reference in such a way that it does not prevent the reference from being garbage-collected.*

References that may be reclaimed by the garbage collector should be classified according to the (garbage-collected) reference types of the host object-oriented language. For example, in JAVA there are both *weak references* for eagerly collected references and *soft references* for references not aggressively reclaimed¹. While weak references are discarded at the next garbage collection cycle, soft references are only collected when the memory is tight.

Cleaning Tasks

A programmer may also require to define customisable cleaning tasks to be automatically executed when a reference is garbage collected. For example, clauses containing dead references may be automatically retracted from the logic database to avoid unexpected behaviours (*e.g.*, null pointer exceptions).

4.3.5 Discussion on Object State Integration Dimensions

In this section we highlight some important points of the integration dimensions reviewed in this section.

Orthogonality of the Dimensions

The different state integration dimensions discussed in this section are orthogonal to each other. For example, associating an object reference with a logic term representation is independent of the opacity of such representation. In other words, this term representation could be either meaningless (*i.e.*, a black box) or meaningful (*i.e.*, a white box) on the logic side, but the mapping of this term with an object reference still should hold.

Another example is the orthogonality of the life span dimension w.r.t. a reification policy. Independently of whether a reference is maintained on the logic side with a symbolic term or a special term wrapping a reference, still the programmer should be able to decide if and when the reference should or should not be garbage collected.

¹<http://docs.oracle.com/javase/7/docs/api/java/lang/ref/Reference.html>

Relevance of Features

Not all the identified dimensions are, of course, relevant in all possible scenarios. For example, dimensions relying on the existence of a garbage collection process only make sense if a garbage collector is available, and can be interacted with, in one of the two languages.

In addition, as figure 4.1 illustrates, features relying on symbolic reification of objects are available in any architecture. Conversely, features relying on a special term wrapping an actual reference are only available if the logic engine is embedded in the object-oriented environment.

Scope of the Inter-Language Conversion Policies

A programmer may need to choose distinct reference management policies in different parts of the program. For example, distinct logic libraries may work better with different representations of the data.

To achieve such a context-dependent representation, a simple mechanism for scoping and encapsulating the best reference handling policy for certain objects is required. Besides greater flexibility, this facilitates performance tuning and testing (*e.g.*, generating mocking representations of references).

4.4 Inter-Language Conversions as a Portable Integration Technique

We claim that by means of symbolic reification most of the advanced techniques available through logic engines embedded in an object-oriented environment can be, to certain extent, emulated with external logic engines. The most difficult task is transparently preserving the identity of an object given an arbitrary logic reification of it.

Instead of relying on the (non-portable) technique of using a special term type wrapping an object reference, we assume an implicit mapping function converting between objects and logic terms.

4.4.1 Specification of Inter-Language Conversion Functions

Inter-language conversion functions act as gateways between representations of an object in different languages. Part of the definition of such a conversion function involves a quantification of its domain and range (*i.e.*, a description of the set of artefacts that the function can accept and return). We make use of the following notation to specify the domain and range of an inter-language conversion function f :

$$f < Domain, Range >.$$

Meaning that the function can convert artefacts belonging to the *Domain* into artefacts belonging to the *Target*, where the *Domain* and *Target* set belong

to different languages. Such domain and range quantifications should be accomplished taking into consideration the natural quantification mechanisms of the two languages taking part in the conversion. In our case, this implies we should be able to quantify over artefacts from both the object-oriented and the logic world.

In many object-oriented languages, the notion of classes plays a major role as a mechanism for describing and categorising objects having similar characteristics [127] (*i.e.*, class instances). Although certain object-oriented languages provide additional mechanisms for quantifying over a set of objects (*e.g.*, *case classes* in SCALA [103] allow using pattern matching over objects), only the notion of classes can be considered as widely spread. Therefore, for the sake of portability we will assume classes as our general mechanism for describing a set of artefacts from the object-oriented world in the domain or range of conversion functions. For example, the domain or range of a conversion function could be specified as all the possible instances of types which are subtypes of **C1** and **C2** (in object-oriented languages not supporting multiple implementation inheritance, **C1** or **C2** may be interfaces).

On the logic programming side, types can also be employed to classify terms. For example, in PROLOG there are predicates (*e.g.*, the **var/1** or **atom/1** predicates) that allow to check the type (*i.e.*, variable, atom, etc.) of a certain term. However, in logic programming it is more natural to quantify over terms by means of a more general term that subsumes them. For example, a term **hello(X)**, where **X** is a logic variable, subsumes **hello('Mary')** or **hello('Peter')**.

From the discussion above, it follows that, depending on the language, there are two portable mechanisms to quantify over artefacts in object-oriented and logic languages: types and term subsumption. While types are applicable to both worlds, term subsumption is applicable only to the logic world.

This implies that, from the point of view of the conversion direction and how their domain and range can be quantified, there are four types of conversion functions:

- f_1 Object-oriented to logic functions whose domain and range are quantified by types. For example, a domain specified as all the subtypes of **C1** and **C2** and a range specified as instances of compound terms.
- f_2 Object-oriented to logic functions whose domain is specified by a type and their range by a general term. For example, a domain specified as all the subtypes of **C1** and **C2** and a range specified as all the terms subsumed by **hello(X)**.
- f_3 Logic to object-oriented functions whose domain and range are quantified by types. For example, a domain specified as instances of compound terms and a range specified as all the subtypes of **C1** and **C2**.
- f_4 Logic to object-oriented functions whose domain is specified by a general term and their range by a type. For example, a domain specified as all

the terms subsumed by `hello(X)` and a range specified as all the subtypes of `C1` and `C2`.

4.4.2 Application of Inter-Language Conversion Functions

Although a programmer should be able to specify a concrete conversion function to apply, in the sake of automation a library accomplishing inter-language conversion operations should define a heuristic for determining which is the right conversion function to apply in different scenarios.

As discussed before, the domain and range of each conversion function defined may include a wide range of artefacts. While a concrete input artefact is fixed at the moment of the conversion, the resulting converted artefact still needs to be inferred.

Unfortunately, we cannot assume that there is always a one to one mapping between artefacts in the two languages. For example, while logic languages typically include a limited amount of primitive types (*e.g.*, integer, float and atom), object-oriented languages deal with more primitive types (*e.g.*, int, long, float, double, boolean, etc). Thus, the right conversion to accomplish is often context dependent (*e.g.*, an atom `true` may be translatable to a string `'true'` or to the boolean `true` according to the purpose of the conversion).

Therefore, a programmer (or a tool helping a programmer) may need to provide a hint to the conversion function on how to accomplish the proper conversion. A convenient mechanism for doing this is to further limit the range of a function to the artefacts expected as the result of the conversion. For example, if the conversion of a logic term is intended to be assigned to the field of an object, then the concrete type of such field could be suggested for guiding the conversion. Given that we have assumed a statically-typed language on the object-oriented side, such field type is available to, for instance, a tool providing high-level integration support.

Since we are using types and terms to define the broad spectrum of artefacts in the range of such functions, it is natural to also use types and terms to further delimit or guide the range of such functions in a concrete conversion operation. Therefore, a conversion function could be parameterised, in addition to the source artefact to convert, with either types (for bidirectional conversions) or terms (for object-oriented to logic conversions), which denote a subset of the range of the conversion function.

Also note that in the case of object-oriented to logic conversions, a programmer should be able to constrain the expected result independently of how the range of the function was specified. For example, a function whose range was specified as all the terms subsuming `hello(X)` is compatible with the constraint that the result of a conversion should be an instance of a compound term. The contrary is also true, a function whose range was specified as instances of compound is compatible with the constraint that the result of a conversion should be subsumed by a term `hello(X)`.

In the next section we provide a formalisation of these ideas.

4.4.3 Formalisation

In this section we formalise inter-language conversion functions quantifying over objects and logic terms by means of their types ($f1$ and $f3$) and functions quantifying over objects by mean of their type and over logic terms by means of a more general term subsuming them ($f2$ and $f4$):

Assuming the following auxiliary functions:

- $ob \alpha T$ is *true* if ob is an instance of T .
- $T \beta ST$ is *true* if ST equals or is a super type of T .
- $term \theta term^*$ is *true* if $term^*$ subsumes $term$ (i.e., $term^*$ is more general than $term$).

And $\overline{T_o}$ being a set of types on the object-oriented side, $\overline{T_t}$ a set of types on the logic side and $term^*$ and arbitrary logic term, we have:

$f_1 < \overline{T_o}, \overline{T_t} > \implies f_1$ can be further delimited as:

$$f_{1_A} : \{o \mid o \alpha T_o^* \wedge T_o^* \in \overline{T_o}\} \times \overline{T_t} \rightarrow$$

$$\{term \mid term \alpha T_t^* \wedge T_t^* \in \overline{T_t}\} :$$

$$f_{1_A}(o, T_t^*) = term$$

and

$$f_{1_B} : \{o \mid o \alpha T_o^* \wedge T_o^* \in \overline{T_o}\} \times \{term^* \mid term^* \alpha T_t^* \wedge T_t^* \in \overline{T_t}\} \rightarrow$$

$$\{term \mid term \theta term^*\} :$$

$$f_{1_B}(o, term^*) = term$$

Description: $f_1 < \overline{T_o}, \overline{T_t} >$ denotes an object-oriented to logic conversion function whose domain and range are specified by means of types.

This function is applicable if the object to convert is an instance of one of the types $\overline{T_o}$ specifying its domain. If the target of a conversion is further constrained (with a second parameter guiding the conversion) by means of a type (f_{1_A}), this function is applicable if such type is an element of the types defining the function's range. If the target of a conversion is further constrained (with a second parameter guiding the conversion) by means of a term (f_{1_B}), this function is applicable if the type of such term is an element of the types defining the function's range.

$f_2 < \overline{T_o}, term^* > \implies f_2$ can be further delimited as:

$$f_{2_A} : \{o \mid o \alpha T_o^* \wedge T_o^* \in \overline{T_o}\} \times \{term^{**} \mid term^{**} \theta term^*\} \rightarrow$$

$$\{term \mid term \theta term^{**}\} :$$

$$f_{2_A}(o, term^*) = term$$

and

$$f_{2_B} : \{o \mid o \alpha T_o^* \wedge T_o^* \in \overline{T_o}\} \times \{st \mid term^* \alpha T_t \wedge T_t \beta st\} \rightarrow$$

$$\{term \mid term \theta term^*\} :$$

$$f_{2_B}(o, st) = term$$

Description: $f_2 < \overline{T_o}, term^* >$ denotes an object-oriented to logic conversion function whose domain is specified by means of types and whose

range by means of a term. This function is applicable if the object to convert is an instance of one of the types \overline{T}_o specifying its domain. If the target of a conversion is further constrained (with a second parameter guiding the conversion) by means of a term (f_{2_A}), this function is applicable if such term is subsumed by the term defining the function's range. If the target of a conversion is further constrained (with a second parameter guiding the conversion) by means of a type (f_{2_B}), this function is applicable if such type is or generalises the type of the term defining the function's range.

$f_3 < \overline{T}_t, \overline{T}_o > \implies f_3$ can be further delimited as:

$$\begin{aligned} f_3 : \{term \mid term \alpha T_t^* \wedge T_t^* \in \overline{T}_t\} \times \overline{T}_o &\rightarrow \\ \{o \mid o \alpha T_o^* \wedge T_o^* \in \overline{T}_o\} : & \\ f_3(term, T_o^*) = o & \end{aligned}$$

Description: $f_3 < \overline{T}_t, \overline{T}_o >$ denotes a logic to object-oriented conversion function whose domain and range are specified by means of types.

This function is applicable if the term to convert is an instance of one of the types \overline{T}_t specifying its domain. If the target of a conversion is further constrained (with a second parameter guiding the conversion) by means of a type, this function is applicable if such type is an element of the types defining the function's range.

$f_4 < term^*, \overline{T}_o > \implies f_4$ can be further delimited as:

$$\begin{aligned} f_4 : \{term \mid term \theta term^*\} \times \overline{T}_o &\rightarrow \{o \mid o \alpha T_o^* \wedge T_o^* \in \overline{T}_o\} : \\ f_4(term, T_o^*) = o & \end{aligned}$$

Description: $f_4 < term^*, \overline{T}_o >$ denotes a logic to object-oriented conversion function whose domain is specified by means of a term and whose range by means of types.

This function is applicable if the term to convert is subsumed by the term $term^*$ defining the function's domain. If the target of a conversion is further constrained (with a second parameter guiding the conversion) by means of a type, this function is applicable if such type is or generalises the types defining the function's range.

These interlanguage conversion functions will provide the basis of our concrete library for inter-language conversions discussed in chapter 6.

4.4.4 Implications and Limitations

At the beginning of this section we discussed that the domain and range of conversion functions can be specified either by means of types or logic terms.

A naive approach to find the most appropriate conversion function would be to traverse all existing functions searching for one whose domain and range are compatible with the source artefact to convert and the intended result. However, this approach will easily impose serious limitations on the amount of

converters that can be registered, since any conversion operation may require to traverse the entire list of available converters. Furthermore, more than one conversion function could be applicable to a conversion operation if two or more converters have overlapping domains and ranges. Therefore, a mechanism for categorising converters in terms of their domains and ranges so that they can be found efficiently and prioritised in relationship to other converters is required.

Given that the domain of conversion functions can be specified using types, a natural way to organise these functions is following the class hierarchy of the object-oriented language. A similar approach is used by the SOUL language. Although in SOUL there is no need for inter-language conversion functions since any object from the host language can be considered as a logic term, it is possible to specify for any class *how* its instances should be unified with a logic term by implementing a special unification method in the class. Given that SOUL runs embedded in a dynamically-typed language, it is possible to add this method (even at runtime) to user or system classes.

The domain of conversion functions can also be specified using terms. In this case, a natural solution could be to assert predicates in a logic engine relating logic terms to conversion functions. These functions can, for example, be wrapped into a special kind of term allowing its inclusion into a logic database. However, part of our portability considerations imply that we should not rely on the logic engine to support special terms wrapping objects. Therefore, we cannot count on a logic engine with such a feature and an alternative should be found.

As we will see in the next chapter, the development of a concrete solution to the problem of how to organise and prioritise conversion functions should be guided by these existing open questions and limitations.

4.4.5 Is the Paradigm Leak Mostly Solved ?

As discussed in section 4.3, integrating object state has been identified in the past as one of the most difficult features to integrate in hybrid systems. Furthermore, part of the complexity of integrating object state is that different strategies for tackling it are available only in certain architectures (*e.g.*, a logic engine embedded in the object-oriented environment).

Unfortunately, from the logic programming perspective there are also concepts that do not have an equivalent in most object-oriented languages (*e.g.*, non-determinism). A discussion of these concepts is provided in the next section.

4.5 Behavioural Integration

In this section we discuss *behavioural* integration aspects both from the logic and the object-oriented programming perspectives. Some of the dimensions identified in this section have been inspired by the ones mentioned by Gybels [66]. Although at first glance some similar mapping problems appear in

both directions of the integration, it is convenient to do a separate analysis of each perspective since there are subtle differences that influence how the problem should be tackled.

Given that we are attempting to achieve an oblivious integration, we would like to design a (semi-)automatic mapping between object-oriented methods and logic predicates. In other words, we attempt that a programmer can invoke a logic predicate from the object-oriented side without having to implement the ‘glue’ code for invoking the predicate and interpreting its result as an object.

Definition 4.13. A *Symbiotic Method* is a method that will be (semi-)automatically delegated to a predicate on the logic side when invoked.

Definition 4.14. A *Symbiotic Class* is a class on the object-oriented side with at least one symbiotic method.

The same is expected from the logic perspective: a programmer should be able to invoke a predicate that (semi-)automatically maps to a method, avoiding to write boilerplate code accomplishing the integration.

Definition 4.15. A *Symbiotic Predicate* is a predicate that will be (semi-)automatically delegated to a method on the object-oriented side when queried.

Definition 4.16. A *Symbiotic Module* is a module or encapsulation unit on the logic side with at least one symbiotic predicate.

In the rest of this section we use S_{mt} to denote the set of all methods on the object-oriented side which are symbiotic. S_c denotes the set of all classes on the object-oriented side which are symbiotic. S_p denotes the set of all predicates on the logic side which are symbiotic. S_{mo} denotes the set of all the modules or encapsulation units on the logic side which are symbiotic.

4.5.1 From the Logic to the Object-Oriented Language

This section discusses the behavioural integration from the logic perspective. A high level view of the mapping from predicates to methods is shown in figure 4.2. The details of this mapping are explained throughout the development of this section.

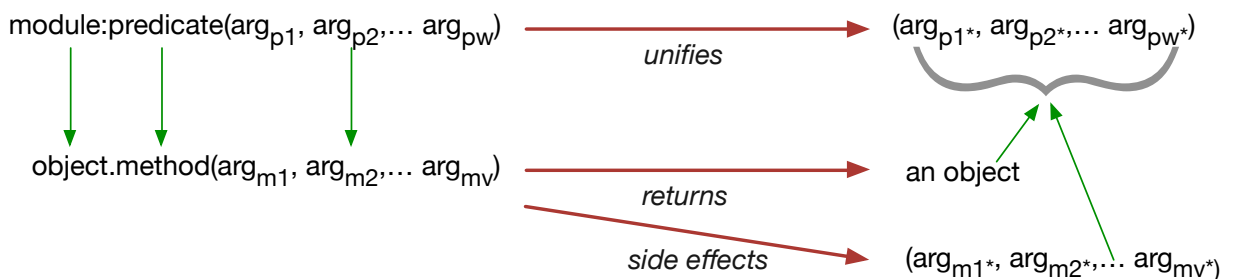


Figure 4.2: Mapping Predicates to Methods.

Inferring the Receiver of the Method

In most object-oriented languages there is typically a receiver for each method invocation. In certain cases, the receiver could be an artefact representing a class of objects. Depending on the language, such class may just be a plain object (*e.g.*, SMALLTALK [60] and RUBY [56]).

Although in logic programming predicates may not necessarily be encapsulated into an entity, often logic languages do provide the notion of a module as an encapsulation unit. Alternatively, object-oriented extensions could provide the notion of an object into the logic programming environment [95, 52, 145, 112].

By assuming one of these encapsulation units, the problem of inferring the receiver of a method is reduced to the problem of mapping a module to such a receiver.

Formally, we define a mapping function τ from modules to objects as:
 $\tau : S_{mo} \rightarrow OB : \tau(mo) = ob$, where $OB = \{ob \mid ob \alpha C \wedge C \in S_c\}$
 which maps a module $mo \in MO$ to a receiver $ob \in OB$ on the object-oriented side (*i.e.*, an instance of a symbiotic class).

Inferring the Name of the Method

A straightforward solution to the problem of inferring a method from a predicate name is to assume they both have the same name. However, a problem with this mapping strategy is the different kind of naming conventions that may exist in the two languages. For example, several object-oriented languages (*e.g.*, JAVA and SMALLTALK) follow a camel-case convention for naming methods (*e.g.*, ‘myMethod’). Conversely, logic languages (like PROLOG) typically name predicates by separating tokens in a name using an underscore (*e.g.*, ‘my_predicate’).

Therefore, introducing method names following a convention distinct to the one expected may limit the transparency of the integration, since the programmer may be forced to work with conventions belonging to two different languages in the same unit.

Note that naming conventions are not just a matter of aesthetics. Certain frameworks and external libraries may depend on the respect of such conventions in order to function as expected (*e.g.*, frameworks based on the JAVABEANS specification [105] or RUBY ON RAILS’ convention over configuration principle [133]).

We believe that it should be up to the programmer to decide on the best policy for mapping routine names between the two worlds. This mapping can be specified by means of a function $v : N_p \rightarrow N_m$
 which maps the name N_p of a predicate $p \in S_p$ to the name of its corresponding method.

Transforming Predicate Parameters to Method Parameters

A mechanism to map predicate parameters (*i.e.*, logic terms) to method parameters should also be specified. Particularly, the types of the parameters in the method should not be limited to being a reification of a logic type (*i.e.*, instances of a class reifying a **Term** class). Instead, any valid type should be acceptable.

To achieve this $\forall p \in S_p$, a programmer should be able to specify a function ϕ such that $\phi(arg_{p1}, arg_{p2}, \dots, arg_{pw}) = (arg_{m1}, arg_{m2}, \dots, arg_{mv})$

Where w is the number of parameters of p and v the number of parameters of the corresponding method. arg_{pi} is a logic term for each $1 \leq i \leq w$ and arg_{mj} is an object for each $1 \leq j \leq v$.

Note that the arity of the predicate parameter list may be different from the one of the method parameters.

In addition, not all predicate arguments may be bound, which brings us to the next integration problem.

Interpreting Unbound Logic Variables

Predicate parameters could include unbound logic variables. Therefore, a semantics for interpreting those variables on the object-oriented side is required. Two scenarios are possible. The first corresponds to mapping a logic variable to a method parameter reifying a logic variable (*e.g.*, an instance of a **Variable** class). In this case, the method should be able to bind a value to the variable, causing a side effect on the logic side. Note that a same variable may appear more than once. If that occurs, binding one variable to a value should affect all of its occurrences. Although this approach allows side effects on the logic side, a disadvantage is that the method on the object-oriented side could have to deal explicitly with the concept of a logic variable, thus making explicit the integration concern.

The other scenario corresponds to the variable being interpreted as an object not reifying a logic term. In that case, an alternative semantics should be defined. For example, assuming the value of the corresponding object as **null** or an equivalent.

Interpreting the Return Value of a Method

The presence of variables in the predicate parameters illustrated a scenario that may cause a side effect after delegating to the object-oriented side (*i.e.*, the binding of the unbound variables). However, the parameters of the method (derived from the predicate parameters) are not the only ones that may determine the new state of the predicate parameters on the logic side. The return value of the method, if present, may also influence the new state of the parameters of the logic predicate.

Therefore, $\forall p \in S_p$ a programmer should be able to define a mapping determining the new value of the predicate arguments:

$$\varphi(arg_{m1*}, arg_{m2*}, \dots arg_{mv*}, r) = (arg_{p1*}, arg_{p2*}, \dots arg_{pw*})$$

Where w is the number of parameters of p and v the number of parameters of the method. arg_{mi*} is the final state of a method parameter for each $1 \leq i \leq v$, r is the method return value and arg_{pj*} is the new state of a predicate parameter for each $1 \leq j \leq w$.

4.5.2 From the Object-Oriented to the Logic Language

In this section we now review the main integration questions that need to be answered from the object-oriented language perspective. Some of them are a trivial inversion of the integration direction from the logic language perspective. Other dimensions are new and solely arise from this new perspective.

Figure 4.3 illustrates a high-level view of the mappings required. All of them are discussed in the remainder of this section.

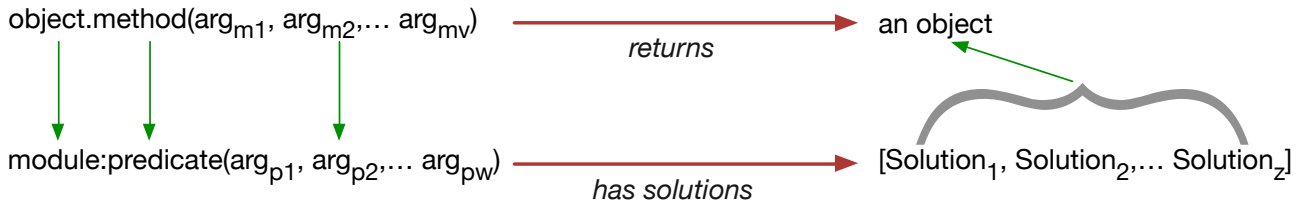


Figure 4.3: Mapping Methods to Predicates.

Inferring the Receiver on the Logic Side

The receiver of a method could be used to infer the module (or object, if an object oriented layer such as LOGTALK is used) on the logic side. Therefore, $\forall C \in S_c$ we can define a mapping function τ^* from all instances of C to modules as:

$$\tau^* : OB \rightarrow S_{MO} : \tau^*(ob) = mo, \text{ where } OB = \{ob \mid ob \alpha C \wedge C \in S_c\}$$

which maps the receiver $ob \in OB$ of a method on the object-oriented side (*i.e.*, an object instance of a symbiotic class) to a symbiotic module $mo \in S_{MO}$ on the logic side.

Inferring the Predicate Name

Predicate names can be inferred from the name of the symbiotic method. This mapping can be specified by means of a function $v^* : N_m \rightarrow N_p$ which maps the name N_m of a method $m \in S_{mt}$ to the name of its corresponding predicate.

Transforming Method Parameters to Predicate Parameters

A mechanism to map method parameters to predicate parameters should be defined. Therefore, $\forall m \in S_{mt}$ a programmer should be able to specify a func-

tion ϕ^* such that $\phi^*(arg_{m1}, arg_{m2}, \dots arg_{mv}) = (arg_{p1}, arg_{p2}, \dots arg_{pw})$

Where v is the number of parameters of m and w the number of parameters of the predicate. arg_{mj} is an object for each $1 \leq j \leq v$ and arg_{pi} is a logic term for each $1 \leq i \leq w$.

Passing Unbound Variables as Predicate Arguments

Logic languages typically allow to pass unbound variables as arguments to predicates. Therefore, the function ϕ^* mapping method parameters to predicate parameters should allow the programmer to specify which predicate parameters are unbound logic variables, or which of them are not ground.

Interpreting a Query Solution as an Object

Since a symbiotic method is interpreted as a query, a mechanism is required for mapping a solution to the query to a return object. A query solution is conceptually a frame f binding values (*i.e.*, logic terms) to variables, such as:

$$f = (Var_1 \rightarrow t_1, Var_1 \rightarrow t_1, \dots Var_n \rightarrow t_n).$$

Hence, $\forall m \in S_{mt}$ the solution to a query is a transformation of this frame to an object by means of a function χ :

$\chi(f, returnType(m)) = ob$, where f is a solution frame and ob its transformation into an object.

Note that the mapping function receives as its second parameter the return type of the symbiotic method to guide the transformation of a query solution to an object. This is an example of the type-guided conversions that were discussed in section 4.4.2.

Non-Determinism Support

One of the major integration issues from the object-oriented perspective is how to deal with non-determinism in the logic side. While methods usually have none or one solution, a logic query may have zero, one, or many solutions. Therefore, $\forall mt \in S_{mt}$ a function ψ mapping query solutions to a single object could be defined as:

$$\psi(f_1, f_2, \dots f_n) = ob$$

where n is an arbitrary number of solutions (possibly zero) and f_i a frame corresponding to a solution i for $1 \leq i \leq n$.

An intuitive description of this mapping is shown in figure 4.4.

4.5.3 Error Handling

Most programming languages provide mechanisms for dealing with exceptional events that alter the normal execution flow of a program. They typically make use of the notion of an *exception* as an artefact capturing information about an anomaly (*e.g.*, an error). When they occur, these exceptions can be captured

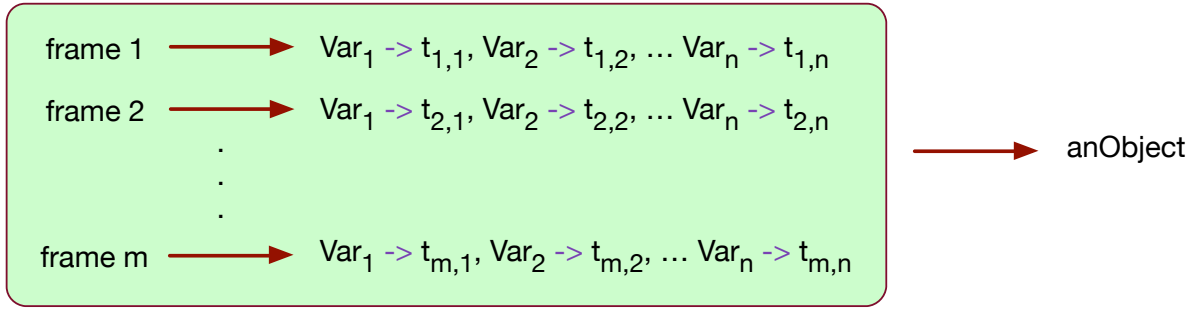


Figure 4.4: From multiple solution frames to an object.

and inspected at well-defined points in a subroutine referred to as an *exception handler*.

Since most object-oriented and logic languages follow this mechanism, there is no strong paradigm mismatch regarding this concept. Instead, if an exception occurs during the execution of a routine in either language, this exception should be propagated to the caller in the other language by means of its natural exception management constructs. However, the exception artefact itself should be translated, as any other artefact, according to the rules discussed in section 4.3.

4.5.4 Discussion on Behavioural Integration

This section has discussed the main behavioural integration dimensions both from the object-oriented and the logic programming perspective.

All the mapping functions declared over a certain artefact (*e.g.*, a module, a predicate, an object or a method) define the artefact's *integration policy*.

Definition 4.17. An *Integration Policy* defines how a *symbiotic artefact* (*e.g.*, a *symbiotic method or predicate*) should be mapped to an equivalent artefact in the foreign language. This policy can be scoped to a certain context.

Although an integration policy exists per each symbiotic predicate and method, a (semi-)automatic integration framework should rely on smart defaults pre-configuring those policies. In this way, the amount of mapping data that actually needs to be provided by a programmer is minimised. Unfortunately, a complete automation (*i.e.*, the lack of a need for custom mappings) is only possible under the assumption that only one mapping between artefacts is possible, which is not the case in many realistic scenarios. For those cases, the programmer should be able to define the mapping functions conforming to an integration policy.

The concrete mechanisms employed to specify integration policies are language dependent. A possible example may be relying on syntactic metadata (*e.g.*, annotations) describing a mapping function between distinct code artefacts. This is the concrete approach explored in the next chapters of this dissertation.

4.6 Chapter Summary

In this chapter, we described our top-down design approach, where the design of an oblivious integration approach is designed first and then refined to the constraints of specific languages and deployment architectures. We identified as the critical integration point, especially regarding portability, the problem of capturing object state from the logic language. We provided a taxonomy of the different dimensions that have to be taken into consideration when tackling the problem of representing state in a logic program. Afterwards, we described and formalised an approach for solving this problem by means of the mapping of inter-language artefacts. Finally we tackled the behavioural integration dimensions, identifying the core integration problems both from the object-oriented and the logic perspective.

5 An Integration Framework Architecture

The shortest answer is doing the thing.

—**Ernest Hemingway**

In this chapter we describe both the high-level and concrete architecture of a framework implementing the conceptual model described in chapter 4. A software architecture intuitively describes the different components of a system, their relations, and the properties of both components and relations [34].

For the design of this architecture we assume JAVA as our object-oriented language and PROLOG as our logic language. As motivated in chapter 1, JAVA guarantees, to a certain degree, the portability and deployability of an application. JAVA also profits from a significant software ecosystem with advanced development tools and a large amount of reusable general-purpose components. PROLOG, on the other hand, facilitates the exploitation of decades of research in logic programming.

5.1 A High Level View

Figure 5.1 provides a high-level view of the components in our integration architecture together with their runtime interactions. Arrows in the figure denote the direction of the information flow from both language perspectives.

In the remainder of this section we discuss the different components of this architecture in detail.

5.1.1 Core Components

For portability reasons, we assume the Java Virtual Machine (JVM) and the Prolog Virtual Machine (PVM) as two separate runtime environments. The main components in these environments are:

- JAVA (1) and PROLOG (7) programs.
- An abstraction of a Prolog Virtual Machine on the Java side (3) and an abstraction of a Java Virtual Machine on the Prolog side (5).

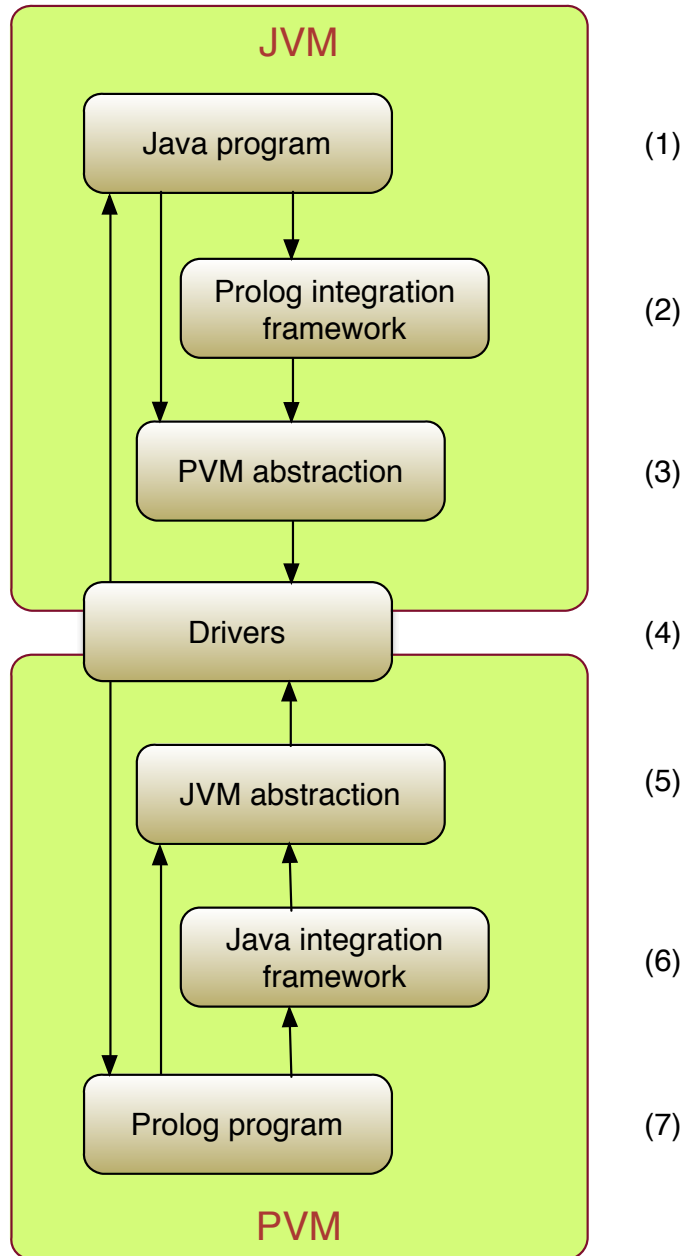


Figure 5.1: Abstract architectural view of our integration framework.

- High level integration frameworks in both environments (2 and 6).
- A set of drivers gluing a Java Virtual Machine with a concrete Prolog engine (4).

We have purposefully adopted a symmetric design, having at an abstract level the same components in both execution environments. In the remainder of this section we discuss the dynamic interactions between these artefacts.

5.1.2 Dynamic Interactions from the Java Perspective

A JAVA program (1) does not interact directly with a PROLOG program. Instead, it interoperates with an abstraction of a Prolog Virtual Machine (3). This

abstraction allows to accomplish both deterministic and non-deterministic logic queries from the Java side. These features will be detailed in chapter 7.

Alternatively, a JAVA program may interact with a higher-level integration framework (2) attempting to provide oblivious integration support. The integration being oblivious in many scenarios, this interaction is not always explicit. This is because the integration is accomplished in a transparent and (semi-) automatic manner therefore avoiding the programmer having to write boilerplate integration code.

In both scenarios (direct interaction with the PVM or through the high-level framework), requests to a logic engine will be delegated to a driver (4) which will delegate these requests in turn to a PROLOG program.

5.1.3 Interoperability Drivers

Drivers are partially implemented in both JAVA and PROLOG. Their objective is hiding from the programmer the complexity of underlying logic engines (*e.g.*, lack of standard features) by providing a common interoperability API to higher-level layers.

5.1.4 Dynamic Interactions from the Prolog Perspective

The perspective from the PROLOG program is, at an abstract level, equivalent to the one for JAVA.

A PROLOG program (7) can either interact directly with an abstraction of a JVM (5) or implicitly by means of a framework providing oblivious integration support (6) with the JAVA-side. As from the JAVA perspective, interaction with the JVM does not occur directly but by means of drivers.

5.1.5 Discussion of the High-Level Architecture

The high-level architecture discussed in this section provides a general overview of which are the main components of our framework and their interactions. However, we have delayed until now a discussion about how to implement the interaction between these components given the paradigm leak existing between object-oriented and logic languages.

In the next section we present a portable approach to reduce this paradigm leak. This will set the ground to introduce our concrete implementation in section 5.3.

5.2 Plumbing the Paradigm Leak with Logtalk

In section 1.1.2 we mentioned that one of the core difficulties of our integration problem was caused by the paradigm mismatch existing between logic and object-oriented languages. While in the object-oriented world we have concepts

such as *objects*, *classes* or *methods*, in the logic world these concepts are replaced by concepts such as *facts*, *rules* or *queries*.

In chapter 4 we observed that a module system on the logic side could help to reduce the paradigm mismatch between the two worlds. Unfortunately, there does not exist a consensual module system for PROLOG [68], despite the existence of an ISO standardisation effort [36].

To compensate for the lack of a common module system, LOGTALK [95] provides a portable object-oriented layer on top of PROLOG. As explained in section 2.6, it provides constructs for implementing several important object-oriented concepts (*e.g.*, structured polymorphism and encapsulation). This reduces the paradigm mismatch (figure 5.2) by modelling the same set of concepts on both sides of the integration.

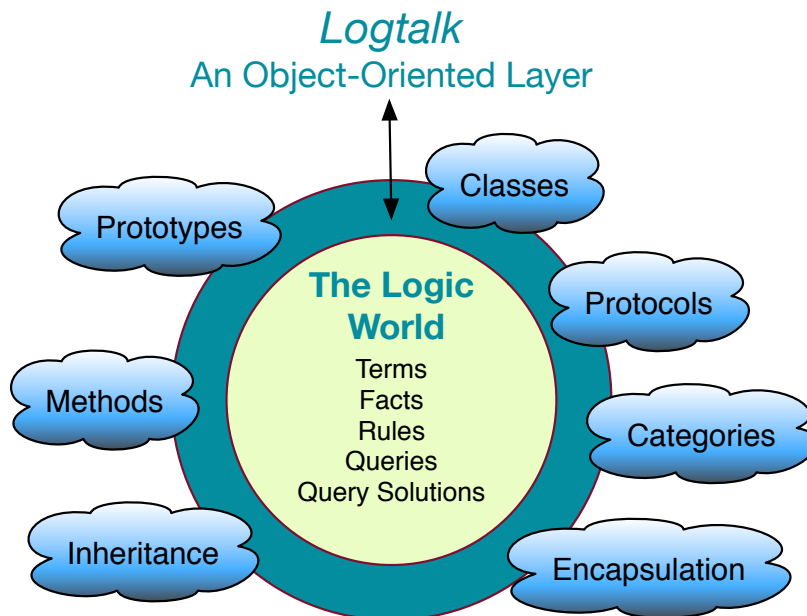


Figure 5.2: Reducing the paradigm mismatch with LOGTALK.

One important reason why LOGTALK helps reducing this mismatch is that the LOGTALK notion of a parameterised receiver of a predicate (*i.e.*, a parametric object [96]) can easily be translated into an object in JAVA. This is because the parameters of such a receiver can be employed to infer the state of an equivalent object on the JAVA-side. This also holds in the other direction. The state of a JAVA object can be employed to determine an equivalent LOGTALK object on the logic side. A detailed explanation of this mapping between JAVA and LOGTALK objects is provided in chapter 6.

In addition to facilitating the mapping of artefacts, another LOGTALK advantage is its advanced support for reflective operations. This allows to easily instrument the invocation of LOGTALK methods. In this way, certain LOGTALK methods can be transparently delegated to JAVA objects. Examples of this are provided in chapter 8.

5.3 The Concrete Architecture of our Integration Framework

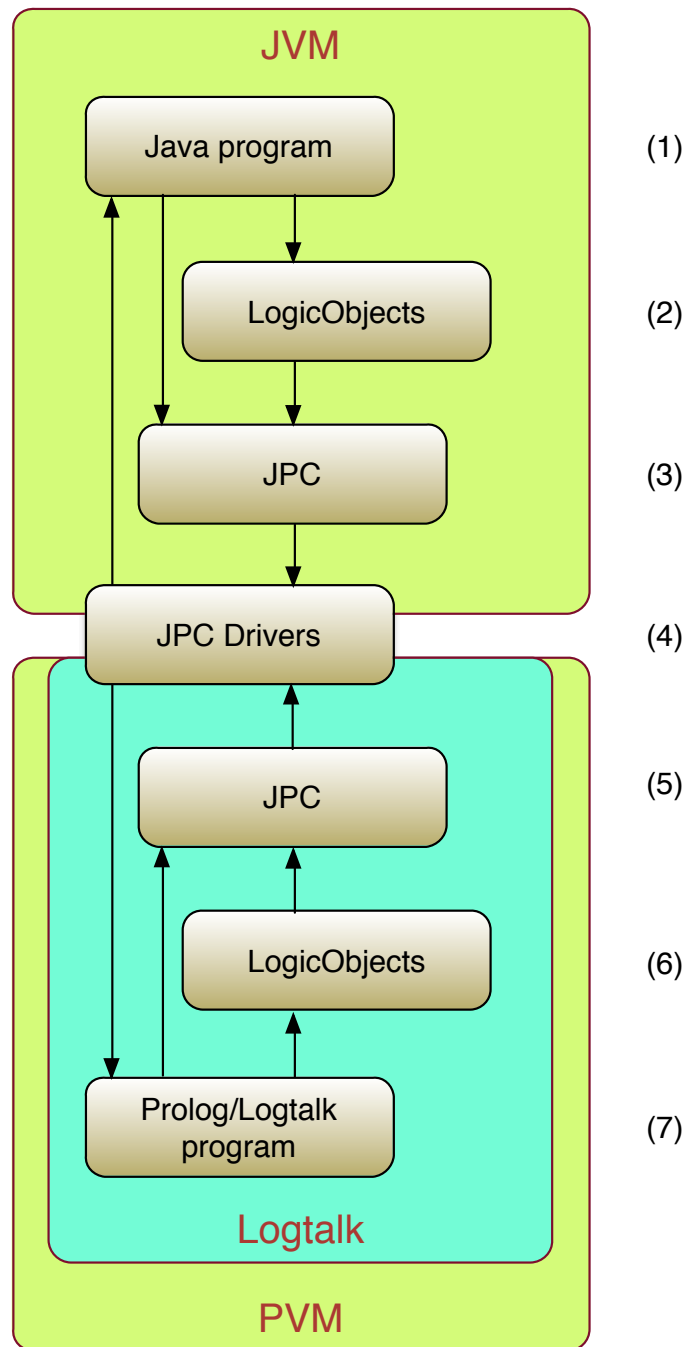


Figure 5.3: Concrete architectural view of our integration framework.

Figure 5.3 illustrates the concrete architecture of our framework. Its most important details are overviewed in this section.

5.3.1 Logtalk Dependency

One of the main differences of our concrete architecture in comparison to figure 5.1 is that we have added LOGTALK on the logic side. On top of it we

have implemented our PROLOG-side artefacts (*e.g.*, the JVM-abstraction and the PROLOG-side driver components).

Although LOGTALK is optional for the integration from the JAVA perspective, the design decision of using LOGTALK implies that our library does depend on it for the PROLOG to JAVA integration direction.

Given the strong focus on portability of LOGTALK, a dependency on it is not a limitation (it is even an advantage) since it is compatible with the great majority of PROLOG implementations which are compliant with official and de facto standards.

5.3.2 The JPC Library

As figure 5.3 illustrates ((3) and (5)), we refer to both the JVM and PVM abstractions as a single library named Java-Prolog Connectivity (JPC) [22]. This library acts as a portable integration layer between JAVA and PROLOG, providing convenient constructs for interacting with a foreign environment.

Foreign Types Reification

Many core features of JPC rely on a reification of the foreign types in the local language. In other words, PROLOG terms are reified as classes in JAVA and JAVA types can be expressed as logic terms in PROLOG.

A discussion about this reification of types, together with JPC's portable interoperability features, will be provided in chapter 7.

Inter-Language Conversions

As part of supporting convenient abstractions for interacting with a foreign environment, JPC provides a convenient API for inter-language conversions between JAVA and PROLOG artefacts. These inter-language conversion features are discussed in chapter 6.

5.3.3 The LogicObjects Library

In our architecture, the high-level library providing an oblivious integration in both languages is named LOGICOBJECTS [23] (*fig.* 5.3 (2) and (6)).

LOGICOBJECTS is an integration framework. Integration frameworks aim to simplify the construction and maintenance of hybrid systems with the goal of reaching an oblivious integration in both the object-oriented and the Prolog world. Techniques may range from the use of aspect-oriented programming for encapsulating integration code and separating it from the main concern of the application [46], to byte code instrumentation and runtime code generation to reduce, or completely eliminate, the need of such integration code [27].

In JAVA, LOGICOBJECTS makes use of runtime code generation and byte code instrumentation [30] to allow a JAVA programmer to interact (semi-)automatically

with PROLOG. In PROLOG, LOGICOBJECTS makes use of the advanced reflective features of LOGTALK for accomplishing an oblivious integration with JAVA.

The LOGICOBJECTS library is discussed in detail in chapter 8.

5.3.4 JPC Drivers

Concrete JPC drivers (*fig. 5.3 (4)*) can be built from scratch, or on top of existing JAVA–PROLOG bridge libraries. In the later case, they may be considered a derivative work of those libraries, possibly implying licensing issues.

Those bridge libraries can be made available by the PROLOG engine provider (*e.g.*, JPL) or by third parties (*e.g.*, PDT CONNECTOR or INTERPROLOG). At the moment, we follow the second approach. Our drivers are currently implemented on top of the JPL, INTERPROLOG, and PDT CONNECTOR libraries. This design decision was ideal for prototyping and as proof of concept. However, performance may be significantly improved with engine drivers specifically designed for JPC.

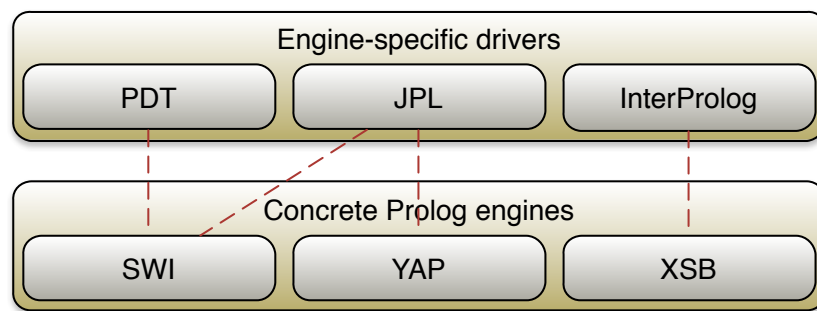


Figure 5.4: JPC drivers.

Figure 5.4 illustrates the current bridge libraries employed to build our JPC drivers. The red lines show the compatibility, at the time of writing, between an existing library and a concrete PROLOG engine.

5.4 Chapter Summary

In this chapter we have discussed the architecture of an integration library providing different levels of integration between an object-oriented and a logic language. At the lowest level, a programmer interacts with the reification, in the local language, of the foreign language environment (*e.g.*, a JVM from the PROLOG-side and a PVM from the JAVA-side). On top of this low-level interoperability layer, a more sophisticated framework allows for transparent and automatic interoperability.

These libraries will be discussed in more detail in the remaining chapters of this dissertation.

6 Inter-language Conversions between Prolog and Java

The original is unfaithful to the translation.

—Jorge Luis Borges

Translation is the art of failure.

—Umberto Eco

In chapter 4 we identified inter-language conversions as a portable mechanism for reasoning over objects on the logic side and interpreting terms as objects on the object-oriented side. We described four types of conversion functions that could be defined by a programmer (section 4.4.1) and argued that it should be possible to scope conversion policies to a user-defined context (section 4.3.5).

In this chapter we introduce JPC: our library which provides a structured mechanism for encapsulating, categorising and applying context-dependent conversions between JAVA objects and PROLOG terms. JPC's inter-language conversion features are inspired on the GSON library [61], which is a library for dealing with context-dependent conversion concerns between JAVA and JSON [39] artefacts.

6.1 Analysis

The illustrative examples of chapter 3 (and in particular the example of section 3.1.2) show the complexity of writing programs that require conversions between artefacts in different languages. At first glance, adding conversion routines directly into the classes whose instances must be converted may look like a straightforward solution. This also has the advantage that such routines are polymorphic (*i.e.*, the conversion to accomplish depends on the actual receiver of the conversion message). However, tangling the main concern of these classes with inter-language conversion routines is a violation of the principle of separation of concerns [48, 107] and hinders the overall maintenance and evolution of a system. Furthermore, this approach is only suitable if the programmer has access to, and can modify, the source code of such classes. Also, it does not provide an appropriate solution for dealing with inter-language conversions

that depend on a particular usage context. This is because with such a solution the conversion would only depend on the callee, but not on the calling context.

There exist techniques that alleviate this separation of concerns problem without requiring access to the source code of the class. A simple approach may be moving the conversion routines into a separate class. While this might work fine for reasonably small systems, a disadvantage is that a programmer cannot execute polymorphic conversion methods, because such methods are no longer part of the class hierarchy of the object to be converted. Therefore, an explicit binding between an object and its conversion routine, located in a different class, should be established (*e.g.*, by means of the double dispatch pattern [59]). Although an ad-hoc infrastructure could be developed to facilitate such mappings (*e.g.*, based on dedicated design patterns or reflection), such infrastructure may not be that trivial to implement and maintain in larger systems.

Aspect-Oriented Programming (AOP) [78] provides an alternative mechanism to encapsulate conversion routines into separate modules (aspects) and weave them into the appropriate classes using a quantification mechanism (*e.g.*, by means of inter-type declarations [86]). This approach achieves a better separation of concerns, but still does not provide a structured solution to the problem of context-dependent inter-language conversions, since with this alternative, that problem is just deferred to the aspect.

We argue that a suitable library for inter-language conversions should provide context-specific constructs for:

- Defining inter-language conversion functions.
- Inferring the best target type of a conversion operation.
- (Optionally) instantiating conversion target types.
- Categorising conversion artefacts.
- Applying conversions.

In addition, an ideal solution should not rely on having access to the source code of classes whose instances participate in conversion operations.

All these properties are discussed in the rest of this section.

6.1.1 Defining Inter-Language Conversion Functions

A programmer should be able to define conversion functions that act as gateways between representations of an object in different languages. In JAVA a conversion function can be specified as a method receiving, among other parameters, the source object to convert and returning the (reified) representation of this object in the target conversion language. Code snippet 6.1 shows a `Converter` interface declaring such a method.

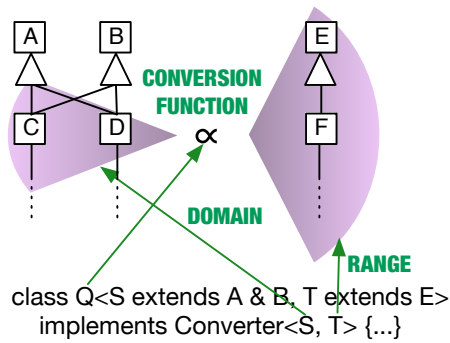


Figure 6.1: Quantifying over variable types.

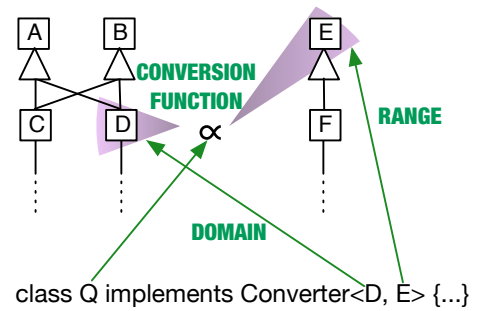


Figure 6.2: Quantifying over concrete types.

```

1 public interface Converter<SourceType,TargetType> {
2     public TargetType apply(SourceType source, Type concreteTargetType, ←
        ...);
3 }

```

Snippet 6.1: The `Converter` interface.

Part of the definition of such a conversion function involves the quantification of its domain and range. Such quantification should be accomplished taking into consideration the natural quantification mechanisms of the two languages taking part in the conversion.

Type-Quantified Converters

Types are a natural JAVA mechanism for quantifying over objects. Therefore, the domain and range of a converter function can be specified by means of types parameterising this function. For example, the type parameters of the `Converter` interface define the domain of the function (`SourceType`) and its range (`TargetType`). These type parameters may reference either a concrete type (parameterised or raw) or a variable type with (explicit or implicit) bounds quantifying over a set of types. For example, at the bottom of figure 6.1 we can see a converter defining its domain and range by means of type variables. Its domain are the classes `C` and `D`, since they inherit from both `A` and `B`, which are the upper bounds of the type variable `S`. Its range is `E` and `F`, according to the upper bounds of the type variable `T`. Note that the upper bounds defining the domain or range of a function can be specified in terms of at most one class and zero or more interfaces. This is because JAVA supports single implementation inheritance and multiple interface inheritance.

As illustrated by figure 6.2, a converter domain and range can also refer to concrete types. In this case, the converter declares as its domain the class `D` and as its range the class `E`.

Term-Quantified Converters

Conversion functions can define concrete PROLOG terms as their domain (*i.e.*, term to object conversions) or range (*i.e.*, object to term conversions). Although terms can also be quantified using (reified JAVA) types (*e.g.*, all instances of class `Compound`), this approach is not ideal since the number of types in PROLOG is quite limited (*e.g.*, specifying the domain of a conversion function as instances of compound terms is, for most cases, too broad). An alternative is to declare what term instances belong to a particular conversion function's domain or range, by means of a more general term that subsumes such instances.

More formally, Sterling and Shapiro [123] define a term A as being an instance of term B if there exists a variable substitution θ such that θ applied to B (denoted as $B\theta$) gives A , so $A = B\theta$. A *substitution* is defined as “a finite set (possibly empty) of pairs of the form $X_i = t_i$, where X_i is a variable and t_i is a term, and $X_i \neq X_j$ for every $i \neq j$, and X_i does not occur in t_j , for any i and j ”.

Based on these definitions, we could, for example, define the domain of a certain converter as all the instances of the term `hello(X)`, where `X` is an unbound PROLOG variable. The terms `hello(mary)` and `hello(peter)` will be included in this domain, while the terms `chello(mary)` or `hello(mary, peter)` will not.

6.1.2 Conversion-Type Inference

As discussed in section 6.1.1, the range of a converter may comprehend more than one type (*e.g.*, if the range is given by a variable type as illustrated by figure 6.1). In those cases, a programmer should be able to select from such range a specific target type for a given conversion operation (hence the `concreteTargetType` parameter in the converter method in code snippet 6.1).

It may also be the case that the converter declares a single generic type as its range, but that such type makes use of variable types (or wildcards) for some of its type parameters (*e.g.*, `List<?>`, where the unbounded wildcard `?` stands for the family of all types) or it is not parameterised (*e.g.*, the raw type `List`). In this case, the programmer may specify a concrete conversion target type (*e.g.*, `List<String>`).

Depending on the converter implementation, type parameters of the conversion target type may influence the result of the conversion. For example, a converter declaring as its range the type `List<?>` may convert a PROLOG list to the parameterised type `List<String>` in a different way than it would convert the same term to the parameterised type `List<Boolean>`.

In case the target type of a conversion is not specified, or is overly general, it should be possible to infer it by inspecting the source object properties. For example, a PROLOG list of compounds with functor `:/2` (*e.g.*, `[a:x,b:y]`) may be typed as a JAVA `Map`. Furthermore, a programmer should be able to customise this (inter-language) type inference mechanism by registering new type inference functions. Such functions can be specified as a method receiving

an object representing an artefact in one of the two languages (*i.e.*, JAVA or PROLOG), and returning the best type for the corresponding object or term in the other language. Code snippet 6.2 shows a `TypeSolver` interface declaring such a method.

Similar to converter functions, the domain of type inference functions is quantified either by types (*e.g.*, the type parameter of the `TypeSolver` interface) or by terms. The range of these functions is implicitly specified as the set of foreign language types (*e.g.*, for JAVA to PROLOG conversions, the range would be the set of classes reifying PROLOG types). Concrete examples are presented in section 6.2.4.

```
1 public interface TypeSolver<T> {  
2     public Type inferType(T object);  
3 }
```

Snippet 6.2: The `TypeSolver` interface.

6.1.3 Factories

Although a converter has access to the (explicit or inferred) target type of a conversion, it may not know how to instantiate this type or how to obtain an existing reference. A simple ad-hoc solution may be to initialise certain converters with a factory (or provider). Then, the converter can delegate to such factory the instantiation of the object to return. However, this is a repetitive problem that may spawn a certain amount of boilerplate code, since converters could become polluted with factory-related code.

A better alternative consists in allowing the programmer to register factory functions thus providing a transparent mechanism for instantiating required objects using registered factories. Similar approaches are provided by dependency injection frameworks such as Google's GUICE library [135]. Factory functions can be specified as a method receiving a type and returning an instantiation of this type. Code snippet 6.3 shows a `Factory` interface declaring such a method. Similar to converter functions, the factory domain is specified by means of a type parameter of the `Factory` interface. Also like converter functions, a factory domain can comprehend more than one type. Therefore, a programmer should be able to specify the concrete type to instantiate (hence the `concreteFactoryType` parameter).

The range of a factory is implicitly specified as the possible set of object instances of the types declared as its domain. We discuss in section 6.2.4 our solution for the management of user-defined factories.

```
1 public interface Factory<T> {  
2     public <T> T instantiate(Type concreteFactoryType);  
3 }
```

Snippet 6.3: The `Factory` interface.

6.1.4 Categorising Conversion Artefacts

The conversion artefacts described before are functions (converter functions, type inference functions, and factory functions). As discussed in section 6.1.1, in certain cases such domains and ranges may be inferred from the declaration of the function. In other cases, they are not implicit in the function declaration and the programmer should explicitly provide them. We argued in section 4.4.4 that conversion functions should be categorised according to their domains and ranges. The same applies to the other kind of functions described before in this chapter. JPC’s categorisation approach is presented in detail in sections 6.2.2 and 6.2.5.

6.1.5 Applying Conversions

From a high-level point of view, the process of applying a conversion consists of the orchestration of the distinct conversion artefacts presented in this section. This process has as input an object to convert, optionally a hint guiding the conversion (*e.g.*, the expected result type) and has as outcome the converted object in a (reified) foreign language representation (*fig.* 6.3).

Given that conversion artefacts (*e.g.*, converter functions) may overlap in their domains or ranges, the implementation of this process should take into consideration delegation and conflict resolution mechanisms. Our approach for the selection of conversions is described in section 6.2.5.

6.1.6 Context as Converter Discriminator

Often the best conversion of a JAVA object to a PROLOG term may depend on the objective of such conversion. For example, a JAVA `List` is better translated to the LOGTALK object `list` if the intention is to invoke a LOGTALK method on that `list` object. Conversely, if the JAVA list would be used as the argument of a predicate, its best term representation is often a PROLOG list.

This is not the only case where the usage context may determine the best conversion. As mentioned before in this section, different converter functions (and other conversion artefacts) may overlap in their domains or ranges. Unfortunately, for those situations it is hard or impossible to conceive a single conflict resolution policy that would best suit all possible cases. However, distinct contexts may discriminate between converters using different policies, thus alleviating this problem.

Hirschfeld et al. [74] defines context as “*any information which is computationally accessible*”. Therefore, any computationally-accessible value should be usable to discriminate converters.

At an abstract level, we define a conversion context as a triple $\text{ctx} = \langle \text{CM}, \text{TIM}, \text{FM} \rangle$ where CM is a *Converter Manager*, TIM is a *Type Inference Manager*, and FM is a *Factory Manager*. A high level view of these components is given by figure 6.3. As their names suggest, the converter manager administers registered converters, the type inference manager administers registered type solvers

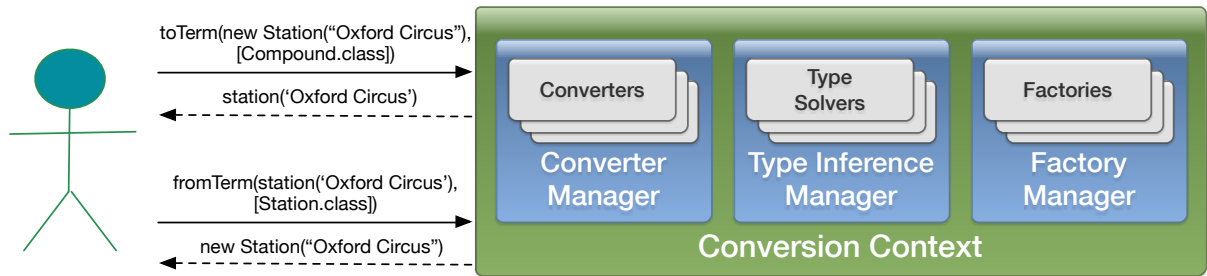


Figure 6.3: The conversion context.

and the factory manager administers registered factories. The concrete implementation of these managers is explained in section 6.2.4.

6.2 Design

In this section we describe the design of the JPC inter-language conversions library.

6.2.1 Design Trade-Offs

The goal of the inter-language conversions library being to facilitate structured conversions between JAVA and PROLOG artefacts, it could have been implemented in either the object-oriented or the logic language. At least, theoretically.

On one hand, being rule-based, this library has a declarative nature. Furthermore, since it needs to reason over the structure of logic terms (the source or target of a conversion operation) a logic language seems to be well suited for this task.

However, the problem is that it also requires to deal with object references. For example, as discussed in section 4.3, the result of applying a conversion from a term to an object could be an existing object reference. Although certain PROLOG implementations support the manipulation of such references, relying on that feature would seriously compromise portability, which is one of our main goals.

Therefore, we have chosen to implement this library entirely on the JAVA-side relying on an embedded PROLOG database that is part of the JPC library. This PROLOG database, although being minimalistic, does support special terms wrapping JAVA object references as required by JPC. This database is described in the next section.

6.2.2 An Embedded Prolog Database

The JPC embedded PROLOG database runs on the JVM and supports the efficient storage of JAVA object references in addition to standard PROLOG terms.

This component is currently not intended to be used directly by the programmer as it lacks many of the features of a full PROLOG system. However, several JPC interoperability features rely on it. Specially, those related to mappings between arbitrary PROLOG terms and JAVA objects (represented as `JRef` terms).

At the moment of writing, its current version supports assertion and retraction of facts, unification, term indexing and backtracking. It does not support rules, operators and many other standard and de facto PROLOG features. Although its current features are enough to support JPC's internal requirements, this engine may evolve in the future towards a standalone embedded PROLOG system.

6.2.3 A Reification of Prolog Terms in Java

As part of our PROLOG engine abstraction, we provide a set of classes reifying PROLOG data types: These classes are:

Term : An abstract PROLOG term.

Atom : A sequence of characters representing a PROLOG atom.

Compound : A compound term consisting of a name and a list of arguments.

IntegerTerm : A PROLOG integer term.

FloatTerm : A PROLOG float term.

Var : A PROLOG variable.

JRef : A JAVA reference term. A special kind of term wrapping a JAVA reference.

6.2.4 The JPC Context

Our primary library class is a conversion context, modelled by the `Jpc` class. This context encapsulates a bidirectional conversion strategy for a set of JAVA objects and PROLOG terms. We will make use of this context to implement, in section 6.3.6, an improved version of the example given in section 3.1.2. The main API of the `Jpc` class is summarised in table 6.1.

In order to facilitate the configuration of a `Jpc` instance, we provide a fluent API for configuring its properties by means of the `JpcBuilder` class. As outlined in section 6.1.6, this configuration involves the registration of converters, type solvers, and factories. For example, code snippet 6.4 shows how to configure a builder to create a `Jpc` context that knows how to convert objects from the example discussed in section 3.1.2. The main API of the `JpcBuilder` class is summarised in table 6.2.

Method	Description
<code>fromTerm(Term)</code>	Converts a term to a Java object
<code>fromTerm(Term, Type)</code>	Converts a term to a Java object instance of a given type
<code>toTerm(Object)</code>	Converts a Java object to a term
<code>toTerm(Object, Class)</code>	Converts a Java object to a term instance of a given term class
<code>inferType(Object)</code>	Returns the inferred type of the object in an inter-language conversion
<code>instantiate(Type)</code>	Returns a new instantiation of the given type

Table 6.1: Main Jpc methods.

Method	Description
<code>register(JpcConverter)</code>	Registers a converter
<code>register(JpcConverter, Term)</code>	Registers a converter associated to a term
<code>register(TypeSolver)</code>	Registers a type solver
<code>register(TypeSolver, Term)</code>	Registers a type solver associated to a term
<code>register(Factory)</code>	Registers a factory

Table 6.2: Main JpcBuilder methods.

```

1 public static final Jpc jpc = JpcBuilder.create()
2   .register(new MetroConverter())
3   .register(new LineConverter())
4   .register(new StationConverter()).build();

```

Snippet 6.4: Building a Jpc context with the JpcBuilder class.

Implementing Converters

The current JPC implementation provides two separate interfaces for defining conversions from JAVA to PROLOG (`ToTermConverter`) and vice versa (`FromTermConverter`). Both interfaces inherit from the common interface `JpcConverter` so they both can be registered into a context with a single call to the method `register(JpcConverter)` introduced before in this section. Behind the curtains, implementors of these interfaces are adapted by our framework, at registration time, as instances of the `Converter` class shown in code snippet 6.1.

The `StationConverter` class in code snippet 6.5 implements both interfaces.

It implements a method defining the conversion of instances of a class `Station` to (lines 6–8) and from (lines 10–13) a PROLOG compound term with the form `station(station_name)`.

```

1 public class StationConverter implements
2     ToTermConverter<Station, Compound>,
3     FromTermConverter<Compound, Station> {
4     public static final String STATION_FUNCTOR = "station";
5
6     @Override public Compound toTerm(Station station, Class<Compound> ←
7         termClass, Jpc context) {
8         return new Compound(STATION_FUNCTOR, asList(new ←
9             Atom(station.getName())));
10
11     @Override public Station fromTerm(Compound term, Type type, Jpc ←
12         context) {
13         String stationName = ((Atom)term.arg(1)).getName();
14         return new Station(stationName);
15     }
16 }

```

Snippet 6.5: The `StationConverter` class.

Implementing Type Solvers

As discussed in section 6.1.2, when no type information is provided in a conversion, our library will attempt to infer the best target type based on the actual source object to convert.

As an example, we mentioned that a PROLOG list term with a certain structure may be reified, by convention, as a map in JAVA. Code snippet 6.6 shows an extract of a type solver applying this rule, which implements the `TypeSolver` interface shown in code snippet 6.2. It returns the `Map` class on line 12 if it can conclude that the term looks like a map. If it is unable to assign a type to the term it signals this by throwing an `UnrecognizedObjectException` exception (line 14).

```

1 public class MapTypeSolver implements TypeSolver<Compound> {
2     @Override public Type inferType(Compound term) {
3         if (term.isList()) {
4             ListTerm list = term.asList();
5             Predicate<Term> isMapEntry = new Predicate<Term>() {
6                 @Override
7                 public boolean apply(Term term) {
8                     return isMapEntry(term);
9                 }
10            };
11            if (!list.isEmpty() && Iterables.all(list, isMapEntry))
12                return Map.class;
13        }
14        throw new UnrecognizedObjectException();
15    }
16
17    private boolean isMapEntry(Term term) {
18        //returns true if term looks like a map entry. false otherwise.
19    }
20 }

```

Snippet 6.6: A type solver for a PROLOG term representing a map.

Implementing Factories

If a converter does not know how to instantiate a conversion target type (*e.g.*, it is abstract), it can ask the Jpc context for an instance of such type. For example, in code snippet 6.6 we show that a PROLOG list with a certain structure will be identified by the type solver as a Map. But the type solver does not provide any mechanism to instantiate such an interface, since its only responsibility is to give a hint on the appropriate conversion type. Assuming that a registered factory can instantiate JAVA maps (code snippet 6.7), a converter only needs to invoke the `instantiate(Map.class)` in a Jpc context to obtain an instance of the desired type.

```

1 public class MapFactory implements Factory<Map<?,?>>() {
2     @Override
3     public Map<?,?> instantiate(Type type) {
4         return new HashMap<>();
5     }
6 };

```

Snippet 6.7: A Map factory.

6.2.5 A Type-Based Categorisation Framework

In section 6.1.1 we discussed that certain conversion artefacts are functions whose domain and range are specified by means of types. This section overviews how our library accomplishes a type-based categorisation of these conversion artefacts. Most examples in this section refer to conversion functions. Type solver and factory functions are subject to a similar categorisation mechanism so we do not repeat the explanation for them.

Source and Target Type Inference

An advantage of a type-based categorisation is that the types defining the domain and range of a conversion function (or other conversion artefacts) can be inferred from their declaration. In code snippet 6.1 we showed that a conversion function is modelled in our framework by means of an interface declaring a method accomplishing the conversion. The first step of the categorisation of a converter consists of inferring the types defining its domain and range (`SourceType` and `TargetType` in code snippet 6.1). This inference is not trivial, since a programmer can provide the actual implementation of this interface (and the concrete type parameters) in a class arbitrarily far from the `Converter` interface in the class hierarchy (*fig.* 6.4). JAVA does not provide a straightforward mechanism for finding the parameter types of a generic class (*e.g.*, the `Converter` interface) given a descendant that defines such types (class B in figure 6.4). Therefore, at registration time we need to traverse the converter class hierarchy until the bindings of its variable types (`Source` and `Target` in figure 6.4) are resolved.

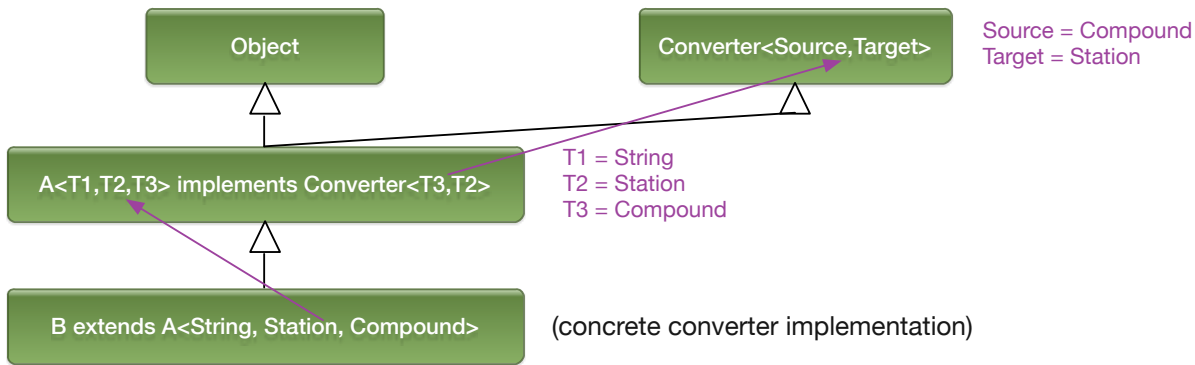


Figure 6.4: Inference of the Source and Target Type of a Converter.

Categorising Converters

Once the domain and range of registered converter functions (or other conversion artefacts) have been inferred, these converters need to be categorised according to these types (section 4.4.4). This section discusses our categorisation approach after discarding simpler alternatives that we took into consideration at the beginning of our design.

A straightforward approach is to consider all registered converters as a single chain of responsibility [59]. Such a chain avoids coupling the sender of a conversion request to a specific converter and allows more than one converter to handle the request. Thus, a converter whose domain and range match the source object and the expected conversion type, can either manage the conversion request themselves or may delegate to the next converter in the chain (*e.g.*, by throwing an exception). A simple conflict resolution policy may be based on assigning weights to converters according to some criteria (*e.g.*, their registration order). However, this approach may result in serious performance penalties in systems with a considerable number of converters.

To solve this performance problem, another approach could be to make use of a map associating classes (the converters domains) to a list of suitable registered converters. Unfortunately, this approach does not provide a solution for managing inheritable converters. A subclass of a class associated with a converter (by means of the map) will not implicitly inherit such converter.

Instead, JPC features a mechanism to associate inheritable dynamic properties (*e.g.*, converters and other conversion artefacts) to classes (*e.g.*, the converter domain types). This allows the categorisation of properties according to a type hierarchy, where these properties are found in a similar way as static class properties (*e.g.*, methods or fields) are resolved. In addition, dynamic class properties are not global, but scoped to a specific JPC context.

In order to illustrate our approach, let's first consider a simplified view of our problem, purposefully ignoring conflicts raised by converters with overlapping domains. In this simplified problem, no JAVA class can be the domain of more than one converter. The more general case is discussed in section 6.2.5.

Figure 6.5 shows an object-oriented hierarchy of animals (example loosely

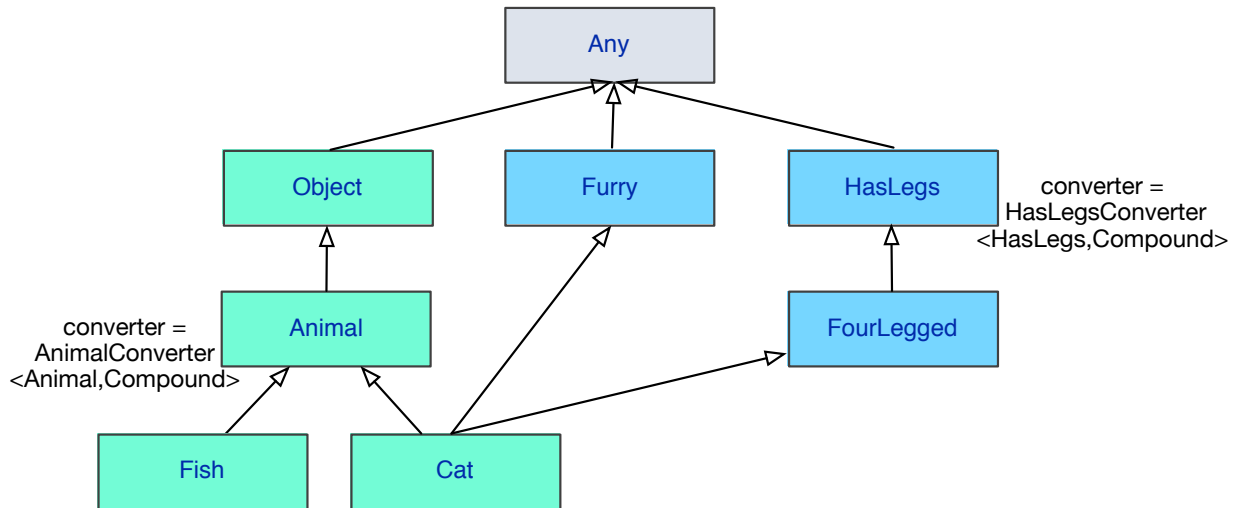


Figure 6.5: A taxonomy of animals and their converters.

based on an example from Odersky et al. [103]). In this hierarchy, **Object**, **Animal**, **Fish** and **Cat** are JAVA classes. **Furry**, **HasLegs** and **FourLegged** are JAVA interfaces. The auxiliary node **Any** denotes the root of the hierarchy.

The figure also illustrates a type-based categorisation of converters according to this class hierarchy (note the *converter* properties associated to certain classes). Observe that our library considers, in many aspects, both classes and interfaces as the same-level categorisation units. For example, one of the two converters shown in the figure (**AnimalConverter**) has the class **Animal** as its domain and **Compound** as its range. The other (**HasLegsConverter**) has the interface **HasLegs** as its domain and **Compound** as its range. This categorisation is automatic, the programmer only has to register a converter as shown in section 6.2.4 and our library will categorise it behind the curtains.

To keep our example simple, the illustrated converters only define one class as their domain. Converters specifying multiple domain classes (as show in figure 6.1) would be associated to all such classes.

Applying Conversions

The previous converter categorisation supports both non-conflictive and conflictive property resolution scenarios. In the simplest case, assume we would like to find the right converter for an instance of **Fish**. Since there is only single inheritance in the type hierarchy of **Fish**, its converter is trivially found from the first ancestor defining such a property (*i.e.*, **Animal**). However, if we are interested in the appropriate converter for instances of **Cat**, distinct property resolution strategies may result in different values. For example, if by convention super classes are looked up first, the appropriate converter is **AnimalConverter**. Conversely, if interfaces are looked up first, the converter would be **HasLegsConverter**.

Although different alternatives exist to solve conflicts for ambiguous properties in multi-inheritance hierarchies, unfortunately there is not a perfect one-

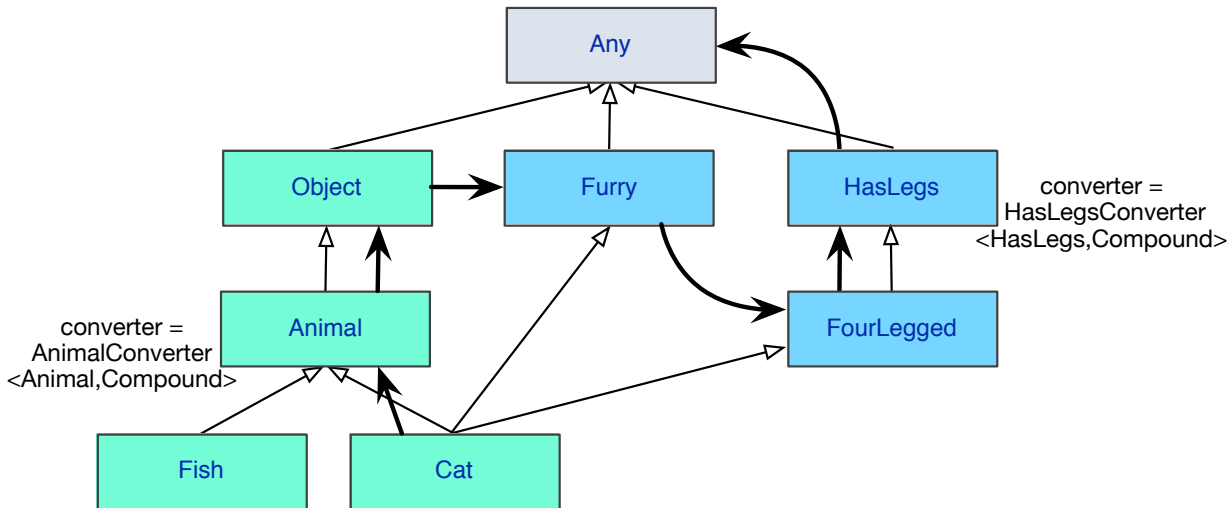


Figure 6.6: Linearization: Left to right (classes first).

fits-all solution [81]. Linearisation helps to solve this conflict unambiguously by defining a linear order in which (super-)categories should be visited when querying a property [49]. In JPC, linearisation is defined as a function mapping a type to a list of types specifying a search path.

The default JPC linearisation function is inspired by the linearisation function used by SCALA [103] to solve conflicts between classes and traits. JPC uses a left-first depth-first search, before eliminating all but the last occurrence of each category in the resulting list (the main difference with the SCALA algorithm is that the later uses a right-first depth-first search). The bold lines in Figure 6.6 illustrate the resolution order of a property starting from the category `Cat`.

The linearisation function first finds this resolution order (left-first depth-first search):

[Cat, Animal, Object, Any, Furry, Any, FourLegged, HasLegs, Any]

which is reduced down to:

[Cat, Animal, Object, Furry, FourLegged, HasLegs, Any] (eliminating all but the last occurrence of a redundant category). As the figure shows, this algorithm has the property that an ancestor category will not be reached until all the descendants leading to it have been explored.

Alternative linearisation functions can be defined on a per context basis (*e.g.*, right to left search, looking at interfaces first) [21]. However, this discussion has been left out of the scope of this dissertation.

Once a converter is found, its range is verified against the conversion target type provided by the programmer. If the target type is in range, the converter is requested to accomplish the conversion. Otherwise, the conversion is delegated to the next converter in the linearisation. A converter may also decide to delegate explicitly to the next converter by throwing a `ConversionException` exception. For example, when attempting to convert an instance of the `Cat` class shown in figure 6.6, the first converter found is the `AnimalConverter` associated

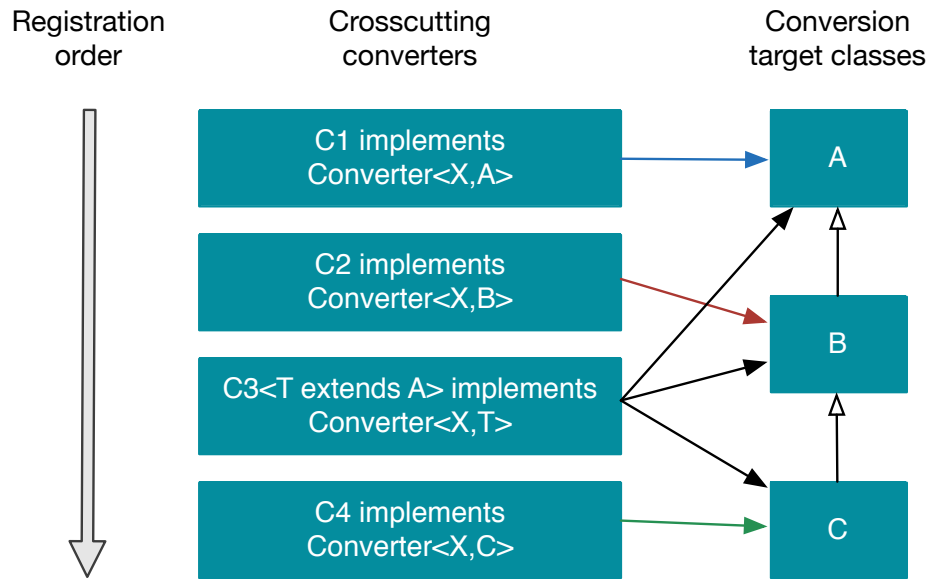


Figure 6.7: Converters with crosscutting domains.

to the `Animal` class. If this converter throws a `ConversionException` exception, the conversion will be delegated to the `HasLegsConverter` converter associated with the `HasLegs` class. If this converter also throws a `ConversionException` exception, it would then be propagated to the caller of the conversion routine, since there are no more converters available in the class hierarchy.

Dealing with Crosscutting Converters

This section refines our problem, assuming that converters may have crosscutting domains. This implies that instead of the `converter` property being a single converter associated to a type (*fig. 6.5*), its value is a composition of (potentially) multiple converters. We consider a composition of converters as a simple converter. Hence, it follows the same rules as our previous example assuming a single converter per class. For example, if a composition of converters cannot perform a conversion operation, it will be delegated to the next composition of converters in the hierarchy (according to the defined linearisation function).

However, we still need to define a mechanism to resolve conflicts between converters in a composition. Figure 6.7 illustrates such problem. We have defined four converters having the same domain (the class `X`) and as range different classes in the class hierarchy on the right. We illustrate our conflict resolution strategy on the scenario of a programmer wanting to convert an instance of `X` to an instance of `B`. Although the actual implemented algorithm may vary a bit for performance reasons, it is reduced to these steps:

- Remove converters whose range is not compatible with the target type of the conversion. This removes the converter `C1`. Although it can convert to instances of `A` (the super class of `B`), we cannot infer from its declaration

that it is able to convert to an instance of B.

- Order the remaining converters according to the ‘distance’ of their range to the target type. This means that a converter declaring as its range the exact target type of the conversion is evaluated before any other declaring as its range a subclass of the target type. In the same way, a converter declaring as its range a direct subclass of the target type is evaluated before anyone declaring as its range a subclass deeper in the hierarchy. According to this criterion, C2 and C3 should have priority over C4.
- In case of remaining collisions (*e.g.*, converters with the same range distance to the target type), order them according to inverse registration order (*i.e.*, converters registered later have more priority than earlier converters). This implies the order C3, C2 and C4 (since C3 is registered after C2). This policy allows programmers to register converters that may override default converters.
- Iterate over the ordered converters attempting to execute the conversion operation. If all available converters have been exhausted, throw a `ConversionException` exception signalling a delegation to the next converter (also a converter composition) in the hierarchy.

Context Dependent Conversions

Let us now put together all the concepts previously described. We illustrate how a context-dependent conversion orchestrates all the conversion artefacts registered into a context.

Figure 6.8 shows a high level overview of this process. Note that this is a simplified view of the actual implementation. Many details (*e.g.*, performance optimisations) have been left out for the sake of conciseness. The figure depicts a conversion from a PROLOG term to its JAVA object representation (the inverse conversion process is similar). First a client of our library invokes the method `fromTerm(Term,Type)` on a JPC context instance (1). The context requests the type inference manager to infer the best conversion type for the term (2). The inference manager iterates over suitable type solvers (3) until one can infer the conversion type of such term (4).

Afterwards, the most specific type between the inferred type (if any) and the type provided by the programmer (if any) is determined (5). Once the best conversion target type has been found, the conversion request is delegated to the converter manager (6). The converter manager attempts to find a suitable converter (7). Once it is found, it is requested to perform the actual conversion (8). If the chosen converter cannot instantiate the conversion target type, it will delegate to the context the instantiation of such type (9). The context will delegate the request to the factory manager (10). The factory manager will attempt to find a suitable factory (11) and in case it is found the instantiation request will be delegated to it. Once a converter succeeds in completing the

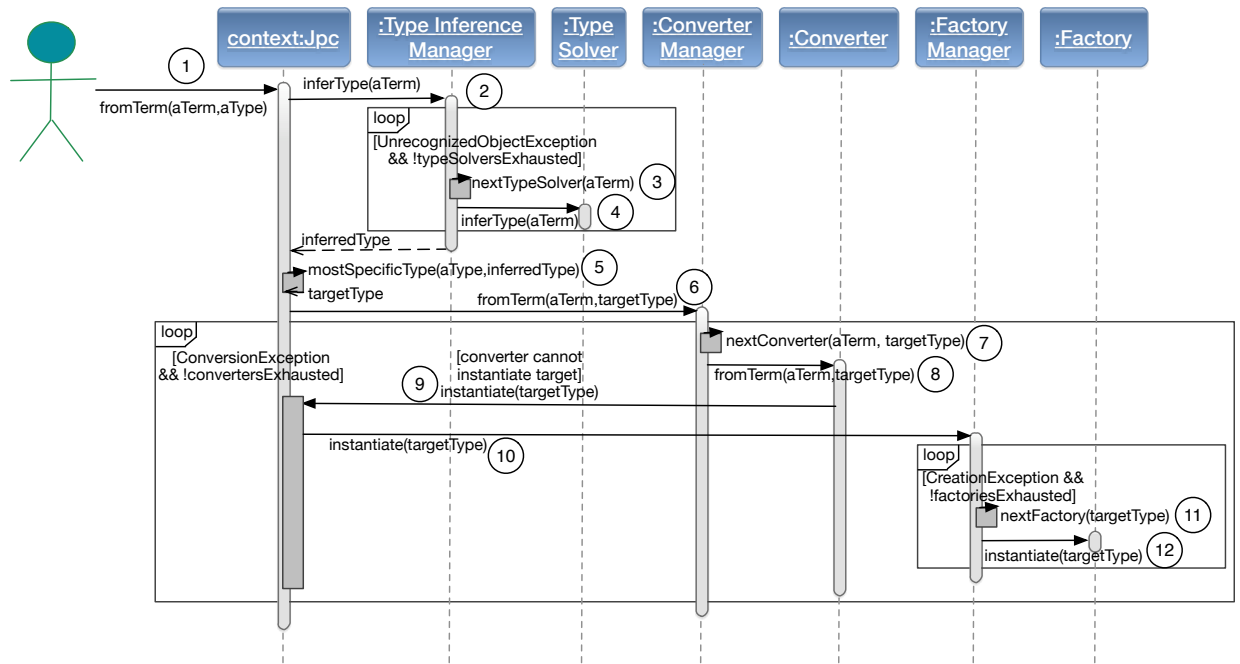


Figure 6.8: A context dependent conversion.

requested conversion, it returns the resulting object to the converter manager. The converter manager returns it to the context instance and the latter to the original invoker.

In the next section we show the features of our library in action by means of concrete examples.

6.3 Inter-Language Conversions

JPC comes with a predefined catalog of converters that support a considerable number of common conversions, to minimise the amount of code to be written when defining new conversions. In the rest of this section we overview both pre-defined and custom conversions.

6.3.1 Primitives Conversions

In this section we illustrate how to convert between JAVA and PROLOG primitives. In order to facilitate the discussion, we consider a JAVA `String` also as a primitive, since it is the natural equivalent of the `atom` primitive PROLOG type.

The simplest way to use our library is by means of the `toTerm(Object)` and `fromTerm(Term)` methods in the `Jpc` class (introduced in section 6.2.4). Code snippet 6.8 shows a list of successful assertions that illustrates some pre-defined conversions of JAVA types to PROLOG terms.

```

1 assertEquals(new Atom("true"),    jpc.toTerm(true));    //Boolean to Atom
2 assertEquals(new Atom("c"),       jpc.toTerm('c'));     //Character to Atom
3 assertEquals(new Atom("1"),       jpc.toTerm("1"));     //String to Atom
4 assertEquals(new IntegerTerm(1),  jpc.toTerm(1));       //Integer to ↵
                                     IntegerTerm

```

```
5 assertEquals(new FloatTerm(1), jpc.toTerm(1D)); //Double to FloatTerm
```

Snippet 6.8: Primitives conversions from JAVA to PROLOG.

Pre-defined conversions of PROLOG terms to JAVA types are shown in code snippet 6.9.

```
1 assertEquals(true, jpc.fromTerm(new Atom("true"))); //Atom to Boolean
2 assertEquals("c", jpc.fromTerm(new Atom("c"))); //Atom to String
3 assertEquals("1", jpc.fromTerm(new Atom("1"))); //Atom to String
4 assertEquals(1L, jpc.fromTerm(new IntegerTerm(1))); //IntegerTerm to Long
5 assertEquals(1D, jpc.fromTerm(new FloatTerm(1))); //FloatTerm to Double
```

Snippet 6.9: Primitives conversions from PROLOG to JAVA.

Note that, f being our default conversion function from a JAVA object to a PROLOG term, and g our default reverse conversion function, it is not always the case that $g(f(x)) = x$, where x is a JAVA primitive object. This is because there are more primitive types in JAVA than in PROLOG. Thus, distinct JAVA objects may be mapped to the same PROLOG term. For example, line 2 of code snippet 6.8 shows that the default conversion of the JAVA character `c` is the PROLOG atom `c`. However, the default conversion of the atom `c` is the String `"c"`. Unfortunately, this is not necessarily always what the programmer expects. The next section describes how to give hints to our library regarding the appropriate conversion to be applied.

6.3.2 Typed Conversions

The `Jpc` class conversion methods can receive as a second parameter the expected type of the converted object. Code snippet 6.10 shows examples of JAVA–PROLOG conversions that specify the expected PROLOG term type. In line 1, the `Integer 1` is converted to an `Atom` instead of an `IntegerTerm` (as in code snippet 6.8, line 4). This is because we specify the `Atom` class as the target conversion type. In line 2, the `String "1"` is converted to an `IntegerTerm`.

```
1 assertEquals(new Atom("1"), jpc.toTerm(1, Atom.class)); ←
  //Integer to Atom
2 assertEquals(new IntegerTerm(1), jpc.toTerm("1", IntegerTerm.class)); ←
  //String to IntegerTerm
```

Snippet 6.10: Typed primitives conversions from JAVA to PROLOG.

In a similar way, code snippet 6.11 shows examples of PROLOG–JAVA conversions that specify the expected JAVA type.

```
1 assertEquals(1, jpc.fromTerm(new Atom("1"), Integer.class)); ←
  //Atom to Integer
2 assertEquals("1", jpc.fromTerm(new IntegerTerm(1), String.class)); ←
  //IntegerTerm to String
3 assertEquals("true", jpc.fromTerm(new Atom("true"), String.class)); ←
  //Atom to String
4 assertEquals('c', jpc.fromTerm(new Atom("c"), Character.class)); ←
  //Atom to Character
```

Snippet 6.11: Typed primitives conversions from PROLOG to JAVA.

6.3.3 Multi-Valued Conversions

The default Jpc catalog of converters also provides conversions for multi-valued data types such as arrays, collections, and maps. Code snippet 6.12 shows a conversion of an array object with a string and an integer element: `["apple", 10]`. Its result is a PROLOG term list having as elements an atom and an integer term: `[apple, 10]`. Alternatively, we could have used a list instead of an array. We would have obtained exactly the same result by replacing line 1 by: `Term term = jpc.toTerm(asList("apple", 10));`

```

1 Term term = jpc.toTerm(new Object[]{"apple", 10});
2 assertEquals(
3   new Compound(".", asList(new Atom("apple"), // equivalent to .(apple, ←
4     new Compound(".", asList(new IntegerTerm(10),
5       new Atom("[]"))))),
6   term);

```

Snippet 6.12: Conversion of an array to a PROLOG term.

A slightly more complex example is illustrated in code snippet 6.13. First, a JAVA map is instantiated (lines 1–5). The default term conversion is applied on line 6, generating a PROLOG list with two key-value pairs: `[apple:10, orange:20]`. This result is tested on lines 8–9.

```

1 Map<String, Integer> map = new LinkedHashMap<String, Integer>() {{
2   // LinkedHashMap preserves insertion order
3   put("apple", 10);
4   put("orange", 20);
5 }};
6 Term term = jpc.toTerm(map);
7 List<Term> listTerm = term.asList(); // converts a Prolog list term to ←
   a list of terms
8 assertEquals(new Compound(":", asList(new Atom("apple"), new ←
   IntegerTerm(10))), listTerm.get(0));
9 assertEquals(new Compound(":", asList(new Atom("orange"), new ←
   IntegerTerm(20))), listTerm.get(1));

```

Snippet 6.13: Conversion of a map to a PROLOG term.

6.3.4 Generic Types Support

Our library provides extensive support for generic types. Consider the example in code snippet 6.14. A PROLOG list term is created on line 1. We use a utility class (from Google's Guava library) to obtain an instance of the parameterised type `List<String>` (line 2). Then we give this type as a hint to the converter (line 3) and we verify on lines 4 and 5 that the elements of the JAVA `List` are indeed instances of `String`, as it was specified on line 3.

```

1 Term listTerm = listTerm(new Atom("1"), new Atom("2"));
2 Type type = new TypeToken<List<String>>().getType();
3 List<String> list = jpc.fromTerm(listTerm, type);
4 assertEquals("1", list.get(0));
5 assertEquals("2", list.get(1));

```

Snippet 6.14: Specifying redundantly the target parameterised type in a conversion.

In the previous example, the type passed to the converter was redundant, since elements in the PROLOG list are atoms, which are converted by default to instances of `String` in JAVA. Consider, however, code snippet 6.15. The main change w.r.t. the previous example is that the type we send as a hint is now `List<Integer>` (line 3). This instructs the converter to instantiate a list where all its elements are integers, as demonstrated on lines 4 and 5.

```

1 Term listTerm = listTerm(new Atom("1"), new Atom("2"));
2 Type type = new TypeToken<List<Integer>>().getType();
3 List<Integer> list = jpc.fromTerm(listTerm, type);
4 assertEquals(1, list.get(0));
5 assertEquals(2, list.get(1));

```

Snippet 6.15: Changing the behaviour of the converter with a parameterised type.

6.3.5 Inference of Conversion Target Types

Code snippet 6.6 showed the implementation of a default type solver responsible of determining if the best conversion type of a term is an instance of `Map`. Code snippet 6.16 shows a conversion example that relies on such type solver. On line 3 we create a list term from two previously created compound terms. We convert it to a JAVA map on line 4 and test its values on lines 5 and 6. As expected, our library infers that the best JAVA type of the term should be a `Map`. This is because the type solver finds that all the elements in the PROLOG list (`[apple-10, orange,20]`) are compounds with an arity of 2 and with functor `'-'`, which are mapped by default to map entries (*i.e.*, instances of the `Map.Entry` class).

```

1 Compound c1 = new Compound("-", asList(new Atom("apple"), new ←
    IntegerTerm(10)));
2 Compound c2 = new Compound("-", asList(new Atom("orange"), new ←
    IntegerTerm(20)));
3 Term listTerm = listTerm(c1, c2); // creates a list term from a list of ←
    terms
4 Map map = jpc.fromTerm(listTerm);
5 assertEquals(10L, map.get("apple"));
6 assertEquals(20L, map.get("orange"));

```

Snippet 6.16: Conversion of a PROLOG term to a map.

Alternatively, line 4 could be replaced by: `List list = jpc.fromTerm(listTerm, List.class)`; This type hint explicitly given by the programmer has higher priority than the one inferred by the type solver. In this case, the result would therefore be a list of map entries since the PROLOG list would be converted to a JAVA list (*i.e.*, an instance of a class implementing `List`), but the default conversion of each term in the list would still be a map entry object.

Note that we leave to the programmer the responsibility of providing enough information (*i.e.*, a target type) in cases where ambiguities are possible (*e.g.*, a term reifying a map). For example, the previous type solver may answer false negatives if it cannot conclude something from the structure of members in the list (*e.g.*, when the list term is empty). If the programmer specifies the intended conversion type the possible ambiguity disappears.

6.3.6 Custom Conversions

A custom converter is created by extending the `ToTermConverter` and `FromTermConverter` interfaces. Code snippet 6.17 shows a custom converter between instances of class `Line` and PROLOG compound terms. The `toTerm()` method converts an instance of class `Line` to a compound term (lines 5–8). The `fromTerm()` method does the opposite (lines 10–16). Although this converter is defining a two-way conversion, a programmer can opt for implement either the `ToTermConverter` or `FromTermConverter` interfaces, defining in this way a unidirectional conversion function. For example, if the problem does not require to convert instances of the `Line` class to a term representation, the `ToTermConverter` interface and the `toTerm()` method declaration may be omitted.

```

1 public class LineConverter implements ToTermConverter<Line, Compound>, ↵
    FromTermConverter<Compound, Line> {
2
3     public static final String LINE_FUNCTOR_NAME = "line";
4
5     @Override
6     public Compound toTerm(Line line, Class<Compound> termClass, Jpc ↵
        context) {
7         return new Compound(LINE_FUNCTOR_NAME, asList(new Atom(line.↵
            getName())));
8     }
9
10    @Override
11    public Line fromTerm(Compound term, Type type, Jpc context) {
12        if(!term.hasFunctor(LINE_FUNCTOR_NAME, 1))
13            throw new ConversionException();
14        String lineName = ((Atom)term.arg(1)).getName();
15        return new Line(lineName);
16    }
17
18 }
```

Snippet 6.17: A Custom JPC Converter

As mentioned in section 6.2.4, a custom converter can be employed to extend a JPC context. An example of a conversion context extended with the `LineConverter` converter is shown in code snippet 6.18.

```

1 Jpc jpc = JpcBuilder.create().registerConverter(new LineConverter()).↵
    build();
```

Snippet 6.18: A Custom JPC Conversion Context

The `StationConverter` and `MetroConverter` converters are defined in the same way as the `LineConverter` converter shown in code snippet 6.17, so we omit their detailed descriptions.

Examples of applications of custom converters are provided in section 7.1.1.

6.3.7 Term-Quantified Converters

In section 6.1.1 we discussed that the domain of PROLOG to JAVA artefact converters is, in some cases, better quantified using terms. Code snippet 6.19 shows a straightforward implementation of the `HelloConverter` mentioned in that section: it returns a JAVA `String` containing the name of the compound to convert, a white space, and the compound first argument (line 3).

```

1 class HelloConverter implements FromTermConverter<Compound, String> {
2     @Override public String fromTerm(Compound term, Type targetType, Jpc ←
        context) {
3         return ((Atom)term.getName()).getName() + " " + ←
            ((Atom)term.arg(1)).getName();
4     }
5 }

```

Snippet 6.19: The *Hello World* converter.

Code snippet 6.20 shows a concrete example of the registration and usage of this converter. In line 3, we make use of the `JpcBuilder` class to register the `HelloConverter` converter. Note that we pass both an instance of the converter and the term `hello(_)` quantifying its domain. This is internally translated to an assert in the embedded JPC PROLOG database, where the `HelloConverter` converter is associated to a domain term with functor `hello/1`. In line 6 we verify that the result of converting the compound term `hello(world)` is the JAVA `String` “hello world”, as specified by the domain quantified converter.

```

1 JpcBuilder builder = JpcBuilder.create();
2 Compound helloCompound = new Compound("hello", asList(Var.ANONYMOUS_VAR));
3 builder.register(new HelloConverter(), helloCompound);
4 Jpc jpc = builder.build();
5 Compound helloWorldCompound = new Compound("hello", asList(new ←
    Atom("world")));
6 assertEquals("hello world", jpc.fromTerm(helloWorldCompound));

```

Snippet 6.20: Applying a term-quantified converter.

6.4 Discussion and Limitations

A limitation of our approach concerns some difficulty regarding the traceability of the converters, inferences and factories employed in a given conversion operation. This may hinder the debugging and maintenance of programs in certain scenarios. Particularly, the frequent usage of converters with crosscutting domains and ranges in multi-inheritance hierarchies may just shift the complexity from writing conversions easily to understanding whether the program

is choosing the proper conversion. These are advanced features that should be managed with care. In most cases, the programmer should favour non-overlapping converters and limiting as much as possible the categorisation of converters in multi-inheritance hierarchies. More inherently complex problems can profit from the flexibility and customisability of JPC's advanced features.

As in the previous discussion, supporting multiple conversion contexts is a powerful feature that should be accompanied with a thoughtful design. Dealing with a great number of such contexts would increase the complexity for a programmer to foresee all their possible interactions, and would make it hard to ensure that programs use the right or expected converters in all possible contexts. Note that the same potential problem is present in popular libraries that also employ the notion of a conversion context (*e.g.*, the GSON library [61]).

6.5 Chapter Summary

This section has introduced a library for accomplishing inter-language artefact conversions between JAVA and PROLOG. Most other features of our approach rely on the API of this library. In the next chapter we describe how JPC allows a programmer to build hybrid JAVA–PROLOG applications.

7 A Portable Context-Dependent Integration Library

Let us make a special effort
to stop communicating with each other,
so we can have some conversation.
—Mark Twain

The previous chapter introduced the JPC’s mechanisms for inter-language conversions between JAVA and PROLOG types. This chapter describes the JPC constructs allowing a JAVA programmer to interact with PROLOG and vice versa.

Before starting JPC’s description, it is important to highlight that this library does not attempt to enable a high degree of automation and transparency (*i.e.*, obliviousness) in the integration. However, it does provide the foundations for the implementation of higher-level libraries. An implementation of such an improved integration library that does try to achieve such automation is described in chapter 8.

7.1 From Java to Prolog

This section describes JPC’s integration features from the JAVA perspective.

7.1.1 Behavioural Integration from Java

Numerous integration libraries rely on the notion of a PROLOG engine as a convenient abstraction for interacting with a PROLOG virtual machine from a JAVA program [14, 130, 129].

Quoting Tarau [14]: “*An engine is a language processor reflected through an API that allows its computations to be controlled interactively*”. Tarau defines a logic engine as “*an engine running a Horn Clause Interpreter with LD-resolution [132] on a given clause database, together with a set of built-in operations*”.

As shown in figure 5.3, JPC too makes use of a PROLOG engine abstraction to facilitate the interaction with PROLOG programs from the JAVA perspective (*e.g.*, by executing queries and inspecting the query results).

Instantiating Prolog Engines

In this section we overview how JPC PROLOG engines can be started and configured. Several techniques and conventions presented here are inspired on APACHE LOG4J [65], a well-known logging library that also makes use of the notion of an engine (a *logging* engine).

In JPC, the easiest way to instantiate a PROLOG engine is by explicitly referencing a JPC driver. As an example, code snippet 7.1 shows how to instantiate an SWI PROLOG engine using the PDT CONNECTOR-based driver mentioned in section 5.3.4 (*fig.* 5.4).

```
1 PrologEngine engine = new PdtConnectorDriver().createPrologEngine();
```

Snippet 7.1: Instantiating a PROLOG engine by explicit referencing a driver.

This approach, although straightforward and good enough for many scenarios, causes a strong coupling between an application and a concrete driver.

Alternatively, JPC allows to easily categorise an *engine space* (*i.e.*, the space of all possible PROLOG engines) according to some developer-chosen criteria.

This approach allows to decouple a JAVA program from concrete engine implementations and provides fine-grained control on the PROLOG engines that should be used to accomplish a given task.

Using this technique, engines are categorised hierarchically following a naming rule. An engine categorised by means of a name will be inherited, unless overridden, by any of its name descendants. For example, in this name hierarchy, a name ‘*org.jpc*’ is said to be the ancestor of the name ‘*org.jpc.Jpc*’. This categorisation is illustrated in figure 7.1. We assume the JAVA naming convention of starting a class name with a capital letter and a package name with a lowercase. Although PROLOG engines can be categorised by means of any name, the JPC convention is to categorise them according to the fully qualified name of the class or package where they are used. The figure shows the names of packages (*i.e.*, ‘*org*’ and ‘*org.jpc*’) and a class name (*i.e.*, ‘*org.jpc.Jpc*’) organised hierarchically. It also shows the effect of categorising a PROLOG engine under the name ‘*org.jpc*’ (*i.e.*, considering this engine also as the default for the class name ‘*org.jpc.Jpc*’). An attempt to categorise a PROLOG engine under an already used category name will result in an exception.

Figure 7.2 shows a more complex engine space with three different PROLOG engines. In this figure, an SWI engine is associated to the package name ‘*com.planning*’ and an XSB engine to the package name ‘*com.routing*’. This means that different engines will be used for classes in these different packages and their sub-packages.

Conversely, a YAP engine is not associated to a package name but to the class named ‘*com.crm.DiscountManager*’. Thus, this engine will not be shared

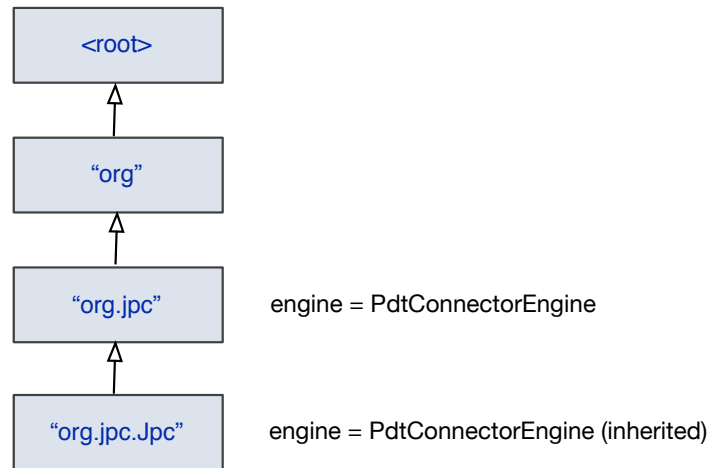


Figure 7.1: A simple JPC engine space.

with other classes implementing PROLOG routines (unless a class is declared as an inner class in `DiscountManager`, since in that case it will have as a name prefix the outer class name).

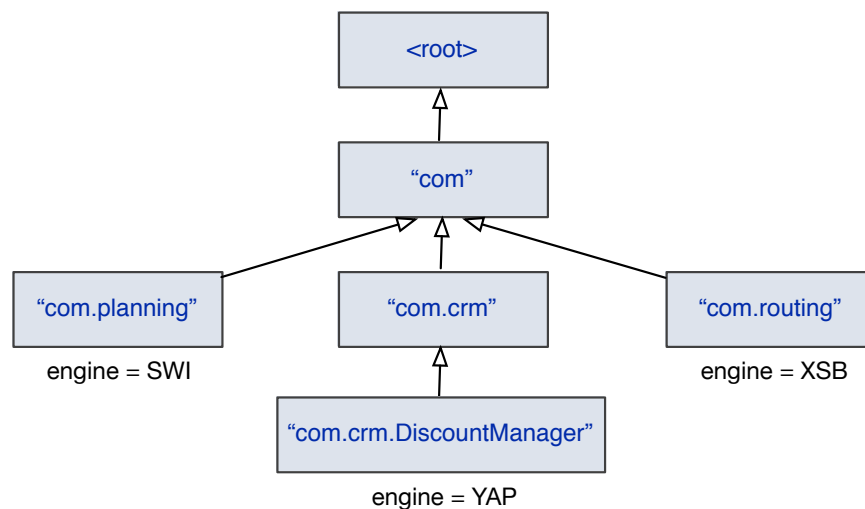


Figure 7.2: A JPC engine space categorising multiple engines.

Such an engine space can be defined either programmatically or by means of a settings file. This configuration consists on the specification of the properties of existing engines, and on the categorisation of these engines according to certain hierarchical names.

A JPC configuration can be represented (*e.g.*, in a settings file) as a JSON object. One of the main attributes of this configuration object is **engines**: a list of engine configurations. As an example, the JPC settings file shown in code snippet 7.2 specifies a LOGTALK-compatible PROLOG engine categorised under the name `'org.jpc'`.

```

1 {
2   "engines": [

```

```

3      {
4          "id": "pdt",
5          "categoryNames": ["org.jpc"],
6          "factoryClass": "org.jpc.engine.pdtconnector.PdtConnectorDriver",
7          "profile": "org.jpc.engine.profile.LogtalkEngineProfile"
8      }
9      ...
10 ]
11 }

```

Snippet 7.2: A JPC settings file.

A detailed discussion of all the attributes in such a JPC settings file has been left out of the scope of this dissertation but can be found in the JPC user manual [22].

Code snippet 7.3 shows how to obtain a PROLOG engine for an arbitrary class.

```

1 import static org.jpc.engine.prolog.PrologEngines.getPrologEngine;
2 ...
3 //returns a Prolog engine categorised for this class fully qualified name
4 getPrologEngine(getClass().getCanonicalName());
5 //or just:
6 getPrologEngine(getClass());

```

Snippet 7.3: Obtaining a PROLOG engine for a given class.

Alternatively, a PROLOG engine may be instantiated by means of the identifier of an engine configuration, as shown in code snippet 7.4.

```

1 import static org.jpc.engine.prolog.PrologEngines.getPrologEngineById;
2 ...
3 getPrologEngineById("pdt");

```

Snippet 7.4: Obtaining a PROLOG engine by its identifier.

By default, engines are instantiated according to the settings file *jpc.settings* in the root package. Alternative mechanisms for configuring engines (*e.g.*, making use of a custom settings file) are explained in the JPC documentation [22].

Query Types

From the definition of a PROLOG engine given by Tarau, we know that we are required at least to be able to specify goals triggering the LD-resolution process. These goals, referred to as queries, may be either:

Deterministic: There is exactly one solution to a query.

Semi-deterministic: There is either one or no solution to a query.

Non-deterministic: There can be any number of solutions, including none.

These distinctions are important from the perspective of a JAVA programmer interacting with PROLOG. For example, a deterministic query may be more performant than using a non-deterministic one for obtaining just the first solution

Query Type	Java Methods	Description
Deterministic	<code>oneSolutionOrThrow()</code>	Returns one solution of the query. If there are no solutions, throws a <code>NoSuchElementException</code> exception.
Semi-deterministic	<code>oneSolution()</code>	Returns an <code>Optional</code> instance, which may wrap one solution or no solution (i.e. an <i>absent</i> <code>Optional</code>) if the query fails.
Non-deterministic	<code>hasNext()</code> <code>next()</code>	Allows to iterate over all the solutions of a query, following the contract of the Java <code>Iterator</code> interface.

Table 7.1: Query Types and their JAVA Method Counterparts

to a goal. This is because deterministic queries may be able to free resources earlier, given that no more solutions are expected.

Finally, in order to facilitate the creation of queries, it should be possible to instantiate them from either a textual or a structured representation.

A textual representation may facilitate the execution of queries that are written by the user of the application (e.g., a PROLOG query browser) or when a programmer knows beforehand the query to submit to a PROLOG engine, maybe requiring some parameterisation. However, if the query is generated at runtime following complex heuristics (e.g., by means of the integration framework that will be discussed in chapter 8) or if security is an issue when working with parameterised queries (e.g., injection of PROLOG code from the user must be prevented), a structured representation may be a better choice. In order to support such structured representation, JAVA classes reifying common PROLOG datatypes should be available. The JPC reification of PROLOG datatypes was already introduced in section 6.2.3. In addition to that, in JPC a PROLOG query is reified as a JAVA class named `Query`. Table 7.1 lists the methods in the `Query` class for starting different kind of logic queries.

The Solution to a Query

At a conceptual level, the solution to a PROLOG query is a frame binding logic variables to PROLOG terms. We represent a query solution as an instance of the class `Solution`. This class implements the `Map` interface. Keys of this map are strings representing variable names. Values are terms bound to these logic variables. Terms bound to variables can be other variables in case a variable was not bound in a solution.

Query Lifecycle

A JPC query has three possible states: **READY**, **OPEN** and **EXHAUSTED**. In this section we describe these states and their transitions.

Figure 7.3 illustrates the transitions between these states by means of a state diagram chart for a JPC query. As shown in the figure, any attempt to invoke a method that is illegal in the current state raises an `InvalidStateException` exception.

The **READY** state denotes that the query has just been initialised or reset. In this state, no message will raise an `InvalidStateException` exception.

The `oneSolution()` or `oneSolutionOrThrow()` methods will attempt to find one solution to the query. If no solution exists, the first method will answer an absent `Optional`, while the second will throw a `NoSuchElementException` exception. In each case, the state of the query is brought back to **READY**.

The `hasNext()` and `next()` methods respect the semantics of the corresponding methods in the `Iterator` interface, which is implemented by the `Query` class. If the query is open, these methods will take the query to the **OPEN** state in case there is at least one solution. If there are no more solutions, the query will be taken to the **EXHAUSTED** state.

The `close()` method is idempotent. Therefore, it can safely be applied several times without changing the state of the query beyond the initial call. The first time it is called, and independently of the current query state, it will take the query to the **READY** state. Any subsequent call to this method while in the **READY** state does not have any further effect on the query life cycle.

The Low-Level Querying API

Code snippet 7.5 shows an example of how to query a **PROLOG** engine using the basic query capabilities of JPC. This code is similar to the one that would be produced when using another bridge library (like the JPL library).

The example shows the implementation of the `lines` method belonging to the `Metro` class introduced in section 3.1.2. This method should return a list with all the lines existing in an underground system.

Lines are represented by the `line/1` LOGTALK object on the logic side and by instances of the `Line` class on the JAVA-side. Line 5 defines a LOGTALK message with the form `line(Line)`, where `Line` is an unbound logic variable. Line 6 creates a goal as the LOGTALK invocation of the previous message on the term representation of a metro instance (*i.e.*, the `metro` atom). The constant `LogtalkConstants.LOGTALK_OPERATOR` refers to the LOGTALK method call operator, which is the atom `::` (section 2.6). Line 7 instantiates a query from this goal. All solutions (a list of `Solution` instances) are requested on line 8. Iterating over these solutions (lines 10–14), a line object is instantiated (line 12) and added to a list of lines (line 13). Finally the list of lines is returned (line 15).

```

1 public class Metro {
2     ...

```

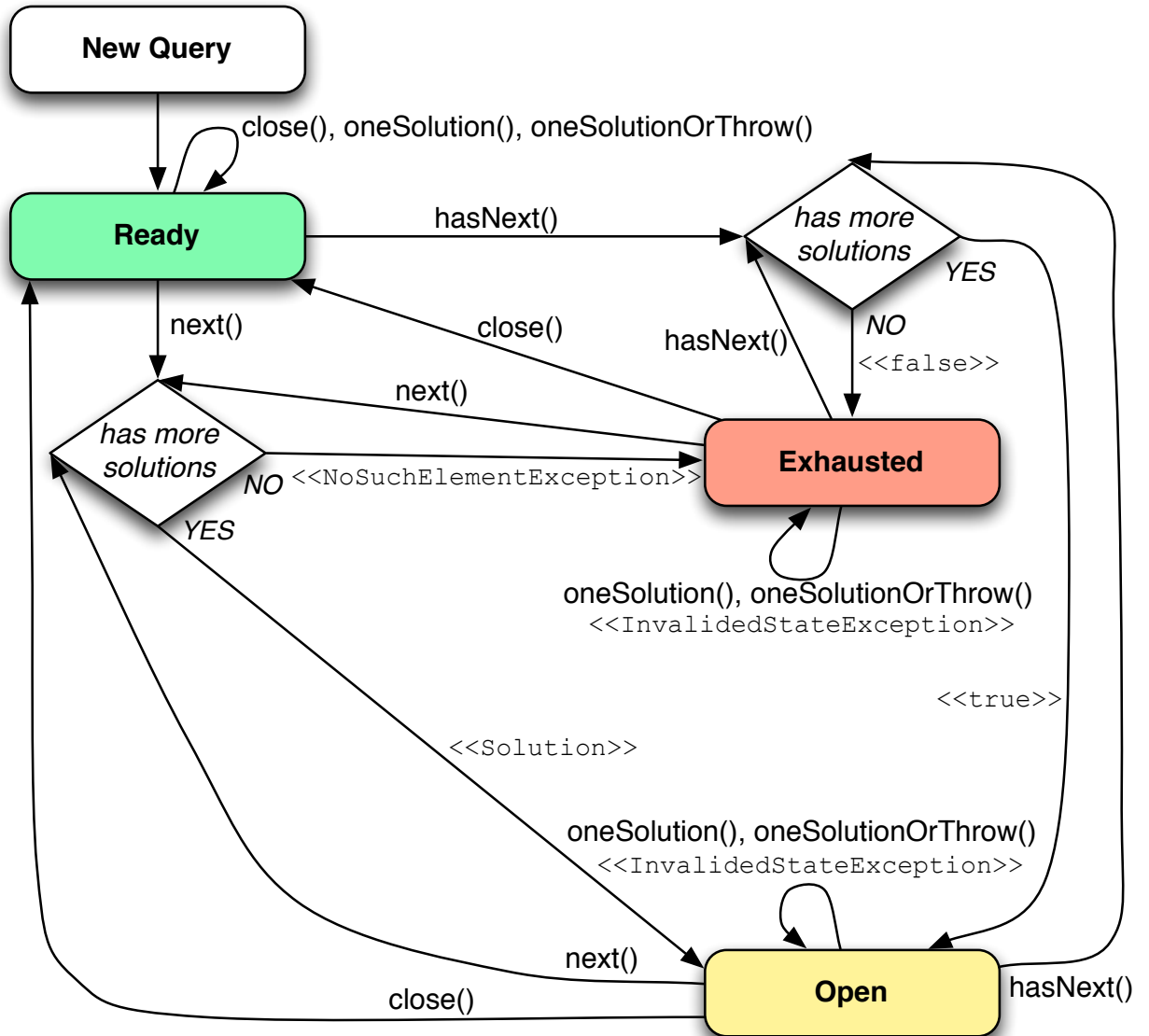


Figure 7.3: State diagram of a JPC query.

```

3 public List<Line> lines() {
4     String lineVarName = "Line";
5     Term message = new Compound(LineConverter.LINE_FUNCTOR_NAME, ←
        asList(new Var(lineVarName)));
6     Term goal = new Compound(LogtalkConstants.LOGTALK_OPERATOR, ←
        asList(asTerm(), message));
7     Query query = prologEngine.query(goal);
8     List<Solution> solutions = query.allSolutions();
9     List<Line> lines = new ArrayList<>();
10    for(Solution solution : solutions) {
11        Atom lineNameTerm = (Atom)solution.get(lineVarName);
12        Line line = new Line(lineNameTerm.getName());
13        lines.add(line);
14    }
15    return lines;
16 }

```

Snippet 7.5: Querying a PROLOG Engine Using the Low Level JPC API.

The High-Level Querying API

In this section we show an alternate implementation of the `lines()` method shown in code snippet 7.5. As discussed in section 6.2.4, programmers can extend JPC by registering custom converters. For this example, we assume a custom conversion context defined as in code snippet 6.4.

In code snippet 7.6, we define the same LOGTALK message as in the previous example (line 4). However, instead of explicitly creating the term representation of a LOGTALK object and a goal term representing the sending of a LOGTALK message, we just instantiate a reification of a LOGTALK object (line 5). Afterwards, on line 6 we request this object to ‘perform’ the message, which returns the reification of a `Query`. Behind the curtains, a term representation of the LOGTALK object is created and a goal with the message sent as an argument is employed to instantiate a query. Line 7 synthesises in one single expression the last 8 lines of the method shown in code snippet 7.5. This is because the operation of (1) iterating over all results of a query, (2) converting each result to a JAVA object representation, and (3) returning all converted objects in a collection, is a common query pattern already encapsulated in JPC.

In this example, the `selectObject()` method returns a query adapter. This adapter converts each query solution to an object representation. How JPC transforms a single query solution to a JAVA object is explained below.

Let’s recall first from section 7.1.1 that a single query solution is a frame binding variables to terms. Taking profit from the fact that JPC provides a simple mechanism for transforming terms to objects, our transformation function receives as input both the query solution to transform and a term representation of the desired JAVA object. This term may contain unbound logic variables. In that case, unbound variables are replaced according to the bindings of the solution. Finally, once the term has been ground, it is transformed to a JAVA object according to the converters registered in the JPC context provided to the query (in this case, the JPC context provided to the LOGTALK object reification).

The term representation of the object to return can be given as an argument to the `selectObject()` method. This term may be given using either its structured or textual representation. If no argument is provided to this method (as in our example), the default term representation is assumed to be the original goal of the query: `line(Line)`. Then, according to our algorithm, each solution of this predicate will be transformed to an instance of the `Line` class. The method `allSolutions()` just facilitates gathering all these objects in a list.

```

1 public class Metro {
2     ...
3     public List<Line> lines() {
4         Term message = new Compound(LineConverter.LINE_FUNCTOR_NAME, ←
5             asList(new Var("Line")));
6         LogtalkObject<Atom> metroLO = new LogtalkObject<>(this, ←
7             prologEngine, jpcContext);
8         Query query = metroLO.perform(message);
9         return query.<Line>selectObject().allSolutions();
10    }
11 }
```

Snippet 7.6: PROLOG engine querying using JPC's automatic conversion features.

As an example where the term representation of the object to return is explicitly given, we revisit the `Station` class of code snippet 3.7. Code snippet 7.7 shows a new implementation of such class. Using our library, the `connected(Line)` method was reduced from 14 to 7 lines of code. In addition, the methods `asTerm()` and `create(Term)` are not in the `Station` class anymore since they have been encapsulated into a converter class. Note that the message term is easily created according to a conversion context (line 6). The last argument of the message term is an instance of `Line`. The conversion of this object to a term is done automatically by our framework. Conversely, in code snippet 3.7 (line 17), we were forced to invoke an explicit conversion when we requested the term representation of the line object.

A `Query` object is instantiated on line 7 from an object abstracting a PROLOG engine. On line 8, the invocation of the `selectObject(String)` method encapsulates the original query in an adapter, where each solution of this query adapter is an object whose term representation is given in the argument of `selectObject()`. In our example, the solution object is expressed as the PROLOG variable `Station`, which has been bound to a term representing an instance of `Station`. The conversion of this term to a `Station` object is transparently accomplished.

```

1 public class Station {
2     ...
3     public Station connected(Line line) {
4         LogtalkObject<Compound> stationLO = new LogtalkObject<>(this, ←
5             prologEngine, jpcContext);
6         String stationVarName = "Station";
7         Term message = jpcContext.toCompound("connected", asList(new Var(←
8             stationVarName), line));
9         Query query = stationLO.perform(message);
10        return query.<Station>selectObject(stationVarName).←
11            oneSolutionOrThrow();
12    }
13 }

```

Snippet 7.7: A JAVA class interacting with PROLOG by means of our library.

7.1.2 Java-Side Reference Management API

This section describes JPC's support for the different dimensions related to the management of JAVA references in PROLOG (figure 4.1).

Symbolic Representation

To illustrate the properties of symbolic references (identified by the first row of figure 4.1), we start by defining a `Person` class (code snippet 7.8) declaring `name` as its only instance variable.

```

1 public class Person implements Serializable {
2     private final String name;
3     public Person(String name) {this.name = name;}
4     ...
5     @Override
6     public boolean equals(Object obj) {
7         ... return ((Person)obj).name.equals(name); //simplified ↵
            implementation
8     }
9 }

```

Snippet 7.8: The `Person` class.

The `PersonConverter` class (code snippet 7.9) defines how instances of class `Person` are translated to a PROLOG compound term (lines 5–7) and back (lines 8–10). According to our classification in section 4.3.2, the term reification of a person, according to this converter, corresponds to a white box representation since it exposes its internal data.

```

1 public class PersonConverter implements FromTermConverter<Compound, ↵
    Person>,
2     ToTermConverter<Person, Compound> {
3     public static final String PERSON_FUNCTOR_NAME = "person";
4
5     @Override public Compound toTerm(Person person, Class<Compound> ↵
        termClass, Jpc context) {
6         return new Compound(PERSON_FUNCTOR_NAME, asList(new Atom(person.↵
            getName())));
7     }
8     @Override public Person fromTerm(Compound personTerm, Type targetType↵
        , Jpc context) {
9         return new Person(((Atom)((Compound)personTerm).arg(1)).getName()↵
            );
10    }
11 }

```

Snippet 7.9: The `PersonConverter` class.

Code snippet 7.10 illustrates a white box term representation of a JAVA object, without object identity preservation (the first three lines are common to most examples; we will not repeat them). A central artefact in our approach is a *conversion context*, instantiated on line 4 using a builder class and configured with the `PersonConverter` converter. With this context we obtain the conversion of a person on line 5 (`person(mary)`). Next, we assert the fact `student(person(mary))` (line 6). A `student(Person)` goal is instantiated on line 7 passing the context defined before. A person is queried on line 8 using a deterministic query. The `selectObject()` method adapts each solution to the query as an object whose term reification is given as a string. This adaptation corresponds to the conversion as a JAVA object of the term that has been bound to the `Person` variable in the solution. Lines 9 and 10 verify that the queried and the original persons are equal, although with different identities.

```

1 final String STUDENT_FUNCTOR_NAME = "student";
2 PrologEngine prologEngine = getPrologEngine();
3 Person mary = new Person("Mary");
4 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
5 Term personTerm = ctx.toTerm(mary);

```

```

6 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm↵
    )));
7 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, ↵
    asList(new Var("Person"))), ctx);
8 Person queriedPerson = query.<Person>selectObject("Person").↵
    oneSolutionOrThrow();
9 assertEquals(mary, queriedPerson);
10 assertFalse(mary == queriedPerson);

```

Snippet 7.10: White Box without Identity Preservation.

Code snippet 7.11 illustrates the mapping of a reference to a term representation (line 2) in the scope of a context. The `newRefTerm()` method associates a person reference (first argument) to an arbitrary (compound) term representation (second argument). In this example, the term corresponds to the term conversion of the reference according to a given conversion context (obtained by the `toTerm()` method of the context instance). We verify on line 6 that this time the queried person corresponds to the original person reference.

```

1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newRefTerm(person, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm↵
    )));
4 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, ↵
    asList(new Var("Person"))), ctx);
5 Person queriedPerson = query.<Person>selectObject("Person").↵
    oneSolutionOrThrow();
6 assertTrue(mary == queriedPerson);

```

Snippet 7.11: White Box and Identity Preservation.

An example of a black box representation is shown in code snippet 7.12. Here, we assert a term of the form `student(serialisation)`, where the compound argument corresponds to the term representation of the serialisation of a `Person` instance. No converter is passed to the query on line 2. This is because the default conversion context (employed by the query if no context is explicitly passed) includes a converter able to deserialize a JAVA object from the term representation of its serialisation. Finally, we verify that our queried person is equal to the original person (line 4) although having different identities (line 5).

Although in the context of this example we have presented this term reification as a black box representation, note that in other contexts this may be considered as a white box. This would be the case if the PROLOG-side is intended to interpret such representation (e.g., if it reasons over the serialised bytes of the object [14]).

```

1 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(↵
    SerializedTerm.serialize(mary))));
2 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, ↵
    asList(new Var("Person"))));
3 Person queriedPerson = query.<Person>selectObject("Person").↵
    oneSolutionOrThrow();
4 assertEquals(mary, queriedPerson);
5 assertFalse(mary == queriedPerson);

```

Snippet 7.12: Black Box without Identity Preservation.

A programmer can also associate an automatically generated term to a reference. An example is given in code snippet 7.13. This time we invoke the method `newRefTerm()` passing as only argument the reference to reify as a term (line 2). A (black box) term representation is generated behind the curtains. Our library guarantees that such generated term representations are identical for the same object even across different contexts.

```

1 Jpc ctx = JpcBuilder.create().build();
2 Term personTerm = ctx.newRefTerm(mary);
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm↵
  )))
4 Query query = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, ↵
  asList(new Var("Person"))), ctx);
5 Person queriedPerson = query.<Person>selectObject("Person").↵
  oneSolutionOrThrow();
6 assertTrue(mary == queriedPerson);

```

Snippet 7.13: Black Box and Identity Preservation.

As discussed in section 4.3.4, a programmer should also be able to control the life span of term-reference mappings. Code snippet 7.14 shows an example. We use the `newRefTerm()` method (line 2) to associate a reference to its (context-dependent) term reification. But afterwards we delete this association using the `forgetRefTerm()` method (line 5). Thus, although the queried person is equal to the original person (line 7) since the term is translated according to the conversion context (line 1), they do not have the same identity (line 8) as the association between the term and the original reference was eliminated.

```

1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newRefTerm(person, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm↵
  )))
4 assertTrue(mary == prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, ↵
  asList(new Var("Person"))), ctx).selectObject("Person").↵
  oneSolutionOrThrow());
5 ctx.forgetRefTerm((Compound)personTerm);
6 Person queriedPerson = prologEngine.query(new Compound(↵
  STUDENT_FUNCTOR_NAME, asList(new Var("Person"))), ctx).<Person>↵
  selectObject("Person").oneSolutionOrThrow();
7 assertEquals(mary, queriedPerson);
8 assertFalse(mary == queriedPerson);

```

Snippet 7.14: Explicit Management of Associations Life Span.

A programmer can also rely on the JAVA garbage collection mechanism for delimiting the life span of an association as shown in code snippet 7.15. The `newWeakRefTerm()` method (line 2) is equivalent to the `newRefTerm()` method discussed earlier. But in this case the association between a term and a reference persists as long as the reference is not reclaimed in the next garbage collection cycle. To prove it, we assign `null` to the only variable keeping a reference to the person (line 4) and give a hint to the garbage collector to start a cycle (line 5). Note that the query is not instantiated with a conversion context (line 7). Thus, an exception is raised when we try to convert the term (bound to the variable `Person`) to an object as no converter is found and no reference is associated to such term. Our framework also provides the `newSoftRefTerm()`

method with similar semantics as `newWeakRefTerm()`, with the only difference that an association between a term and a reference may persist some time after a garbage collection cycle, and will be deleted only if the memory gets tight.

```

1 Jpc ctx = JpcBuilder.create().register(new PersonConverter()).build();
2 Term personTerm = ctx.newWeakRefTerm(mary, ctx.<Compound>toTerm(mary));
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(personTerm←
    )));
4 mary = null;
5 System.gc();
6 try {
7     prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(new Var(←
        "Person")))).<Person>selectObject("Person").oneSolutionOrThrow()←
        ;
8     fail();
9 } catch(ConversionException e) {}

```

Snippet 7.15: Garbage Collection Management of Associations Life Span.

Object Reference Representation

This section focuses on the properties of object references (identified by the second row of figure 4.1). Although our library currently only has drivers for non-embedded PROLOG engines, as a proof of concept we implement the examples in this section using the JPC embedded PROLOG database described in section 6.2.2. With the exception of open unification, all the other properties are supported by our implementation.

We start with an example of constant unification of references in code snippet 7.16. As mentioned in section 6.2.3, a JPC `JRef` term wraps an object reference. In our current version, they are unified as constants (i.e., unifying two `JRef` terms succeeds if their referred objects are equal). In line 1 we assert that *mary* (wrapped in a `JRef` term) is a student. In line 2 we query if a different person object with the same name is a student, which succeeds.

```

1 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.jRef(←
    mary))));
2 assertTrue(prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(←
    JRef.jRef(new Person("mary")))).hasSolution());
3 Solution solution = prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, ←
    asList(new Var("X")))).oneSolutionOrThrow();
4 JRef<Person> jRef = (JRef<Person>) solution.get("X");
5 assertTrue(mary == jRef.getReferent());

```

Snippet 7.16: Constant Unification of `JRef` terms.

Thanks to our embedded PROLOG database, the identity of a reference is trivially preserved. To illustrate this, we execute a deterministic query (line 3) with goal `student(X)`. We verify that the obtained referent has the same identity as *mary* on line 5.

Code snippet 7.17 shows how to create `JRef` instances that may be garbage collected. We first create two objects equal to *mary* and assert them, using two kind of references: *strong* (line 3) and *weak* (line 4). When we query for students unifying with *mary* (line 5) using a strong reference, we get two results instead

of one. This is because the unification semantics of `JRef` terms evaluates the referents (a person named *‘mary’*), not the actual `JRef` term wrapper stored in the PROLOG database (*i.e.*, a weak, soft or strong term reference wrapper). Afterwards we assign to `null` the variable `person2` (line 6) and give a hint to the garbage collector to execute a cycle (line 7). Since the referent of the `JRef` term asserted on line 4 has been invalidated, the number of students unifying with *mary* is now only 1 (line 8).

Note that weak or soft references should be used with care: they may require non-monotonic reasoning as the referent of a `JRef` term may be invalidated during the query execution.

```

1 Person person2 = new Person("Mary");
2 Person person3 = new Person("Mary");
3 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.jRef(↵
    mary))));
4 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.↵
    weakJRef(person2))));
5 assertEquals(2, prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, ↵
    asList(JRef.jRef(mary)))).allSolutions().size());
6 person2 = null;
7 System.gc();
8 assertEquals(1, prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, ↵
    asList(JRef.jRef(mary)))).allSolutions().size());

```

Snippet 7.17: Life Span of `JRef` terms.

The previous example motivates the need of a cleaning mechanism. Code snippet 7.18 illustrates such mechanism using a user-defined cleaning task. To keep our example simple, this cleaning task retracts all the asserted students (lines 1–5) when a reference is invalidated. A more sophisticated example would retract only the invalidated reference. Our cleaning task is associated with a weak reference on line 6. In line 9 we verify that no students are in the database after the reference to *mary* has been invalidated (lines 7–8).

```

1 Runnable cleaningTask = new Runnable() {
2     @Override public void run() {
3         prologEngine.retractAll(new Compound(STUDENT_FUNCTOR_NAME, asList(↵
            (Var.ANONYMOUS_VAR))));
4     }
5 };
6 prologEngine.assertz(new Compound(STUDENT_FUNCTOR_NAME, asList(JRef.↵
    weakJRef(mary, cleaningTask))));
7 mary = null;
8 System.gc();
9 assertFalse(prologEngine.query(new Compound(STUDENT_FUNCTOR_NAME, asList(↵
    Var.ANONYMOUS_VAR))).hasSolution());

```

Snippet 7.18: Cleaning Tasks.

7.2 From Prolog to Java

This section describes JPC’s integration features from the PROLOG perspective.

7.2.1 Behavioural Integration from Prolog

On the logic side, the `java/0` LOGTALK object is the central interaction point with a Java Virtual Machine. The public methods of this object are summarised below.

eval(Expression, ReturnSpecifier) Evaluates in JAVA the expression `Expression`. Afterwards it grounds `ReturnSpecifier` according to the returning value.

eval(Expression) Same as `eval/2` but ignores the expression's return value.

invoke(Expression, Message, ReturnSpecifier) Sends a message in JAVA to the object resulting of evaluating the expression `Expression`. The message is expressed as an atom or compound. Afterwards it grounds `ReturnSpecifier` according to the returning value.

invoke(Expression, Message) Same as `invoke/3` but ignores the method return value.

get_field(Expression, FieldName, ReturnSpecifier) Gets the field with name `FieldName` of the object resulting from the evaluation of the expression `Expression`. Afterwards it grounds `ReturnSpecifier` according to the field value.

set_field(Expression, FieldName, FieldValue) Sets the field with name `FieldName` of the object resulting from the evaluation of the expression `Expression` according to the value of `FieldValue`.

forget(Reference) Forgets the mapping of `Reference` to an object.

Support for Object-Oriented Expressions

Most of the public methods of the `java/0` LOGTALK object require a term representing an object-oriented expression. JPC makes use of a loose definition of what such an object-oriented expression is: it can be as simple as an object reference or a method invocation. However, nothing prevents to define term equivalences for any possible construct in the object-oriented language [80].

JPC is in general agnostic of concrete equivalences between term expressions and object-oriented constructs. This is an intentional design decision that has as objective to allow different higher-level integration frameworks to define the equivalences that are best suited for them. These frameworks can accomplish that by registering bidirectional converters in the appropriate conversion context as discussed in chapter 6.

As an illustration of this, we will provide in chapter 8 concrete equivalences between terms and object-oriented expressions that are employed by LOGICOBJECTS, a higher-level interoperability framework developed on top of JPC.

Return Specifiers

In section 7.1.2 we discussed how a JAVA programmer can explicitly customise an object reification policy. In this section we describe how the same can be accomplished from the perspective of the PROLOG programmer.

Most methods provided by the LOGTALK `java/0` object refer to a return specifier. A return specifier is a compound providing information about how an object returned from the JAVA-side should be interpreted in PROLOG. This compound, as part of its arguments, also includes the return value itself. This can be an unbound variable bound to the return value upon evaluation. Alternatively, it can be a non variable term. In that case, the predicate will succeed only if the provided return value unifies with the actual returned value.

The possible return specifiers are described below. In all cases, the `Result` variable is the term representation of the actual value returned from the JAVA-side.

term(`Result`) `Result` is the object returned reified as a term. No relation between the term representation and the original reference is established.

jref(`Result`) `Result` is a generated term identifying the returned object. The mapping between this term and the object persists as long as the term exists in the PROLOG engine. This demands a JPC driver supporting this feature.

strong(jref(`Result`)) `Result` is a generated term identifying the returned object. The mapping between this term and the object persists as long as it is not explicitly dropped with a call to `java::forget(Result)`.

weak(jref(`Result`)) `Result` is a generated term identifying the returned object. The mapping between this term and the object is guaranteed to persist as long as the object is referenced on the JAVA-side. Once the object is not referenced, the mapping will be dropped in the next JAVA garbage-collection cycle.

soft(jref(`Result`)) `Result` is a generated term identifying the returned object. The mapping between this term and the object is guaranteed to persist as long as the object is referenced on the JAVA-side. Once the object is not referenced, the mapping will be dropped when the available memory gets tight.

strong(jref_term(`Result`)) Same as `strong(jref(Result))`, but the term representation of the object is the one configured according to the default conversion context.

weak(jref_term(`Result`)) Same as `weak(jref(Result))`, but the term representation of the object is the one configured according to the default conversion context.

soft(jref_term(Result)) Same as **soft(jref(Result))**, but the term representation of the object is the one configured according to the default conversion context.

serialized(Result) **Result** is a term encapsulating the object serialised data. The relation between the original object and its term representation is lost (*i.e.*, transforming the term back to an object will produce an object equals, but not identical, to the original object).

The default conversion context mentioned above is an instance of the class `org.jpc.Jpc`. In the current implementation, this context can be accessed and configured with the static methods `getDefault()` and `setDefault()` in the `Jpc` class.

Usage examples of return specifiers will be given in chapter 9.

Reifying Java Types in Prolog

JPC allows a PROLOG programmer to express JAVA types as logic terms. As we will see, this reification of types allows to customise the conversion of terms to objects. It also facilitates the instantiation of new objects and the execution of static methods.

A complexity in reifying JAVA types for our purposes is the intrinsic dichotomy existing in JAVA between what is the reification of a class (*i.e.*, an instance of the class `java.lang.Class`) and what a class as a linguistic syntactic element means. For example, a term representation of an instance of `java.lang.Class` could be used to invoke any of the reflective methods defined in such class (*e.g.*, `getDeclaredField()`). Conversely, the term representation of a class as a linguistic syntactic element attempts to abstract a class as an object having as its fields the static variables declared in the JAVA class and as its methods the declared static methods.

Unfortunately, it is not convenient to use a single representation for these two concepts, because there would be no way to distinguish between methods declared in the `java.lang.Class` from static methods in the class having the same name. Therefore, JPC makes a distinction between these two different JAVA concepts and distinct term representations are used to refer to one or to the other.

The terms reifying JAVA types are described below.

class(Packages, ClassNames) is interpreted as a JAVA class syntactic element (*e.g.*, it understands static methods or the `new` message keyword).

Packages is a (possibly empty) list of package tokens. **ClassNames** is a non-empty list of class names. For example, the term `class([p1,p2,p3], [c1,c2])` is interpreted as the class `c2` declared as a member of the class `c1`. The package of `c1` is `p1.p2.p3`.

class(FullyQualifiedName) A shorter convenient notation for `class/2`.

FullyQualifiedName is the fully qualified name of the JAVA class. For

example, `class('p1.p2.p3.c1$c2')` is equivalent to `class([p1,p2,p3], [c1,c2])`.

type(Packages, ClassNames) Its arguments are the same as specified for `class` /2, but it refers to the object reifying the class in JAVA (an instance of `java.lang.Class`). For example, the term `type([p1,p2,p3], [c1,c2])` is interpreted as an instance of `java.lang.Class` reifying the class `c2` declared as a member of the class `c1`. The package of `c1` is `p1.p2.p3`.

type(FullyQualifiedName) A shorter convenient notation for `type`/2. `FullyQualifiedName` is the fully qualified name of the JAVA class. Therefore, `type('p1.p2.p3.c1$c2')` is equivalent to `type([p1,p2,p3], [c1,c2])`.

array(ComponentType) An array type. `ComponentType` is an arbitrary type, possibly another array type. For example, the type `array(array(type('java.lang.String')))` represents an instance of `java.lang.Class` reifying a bi-dimensional JAVA array of the class `String`.

type(PackageNames, ClassNames, TypeArguments, OwnerType) A type with parameters. The first two arguments are the same as `type`/2. `TypeArguments` is a list of type arguments. `OwnerType` is the type object representing the type that this type is a member of. For example, if the type is `0<T>.I<S>`, the owner type is a representation of `0<T>`. As an example of the term reification of a parameterised type, the term `type([java,util], ['List'], [type([java,lang], ['String'])], _)` reifies the type `java.util.List<java.lang.String>`. The last argument, an unbound PROLOG variable, is interpreted as a JAVA `null`.

variable_type(Name, GenericDeclaration, UpperBounds) A type variable. `Name` is the variable name. `GenericDeclaration` the generic declaration declared for this type variable (*e.g.*, a class). `UpperBounds` is a list with the variable upper bounds. For example, `variable_type('E', type('java.util.List'), [type('java.lang.Object')])` represents a type variable `E` declared in the class `java.util.List` and having as its sole upper bound `java.lang.Object` (the default upper bound).

variable_type(UpperBounds, LowerBounds) A wildcard type. `UpperBounds` and `LowerBounds` are the upper and lower bounds of the wildcard type. For example, `variable_type([], [type('java.util.ArrayList')])` represents a wildcard type with no upper bounds and as its lower bound the class `java.util.ArrayList`.

Customising Conversions from Prolog to Java

JPC provides a catalog of common conversions between logic terms and JAVA artefacts. Some of these default conversions were discussed in chapter 6.

In this section we discuss how a PROLOG programmer can bypass the default JPC conversion mechanism or give hints to it.

JPC makes two techniques available, both relying on wrapping the term to convert into another term interpreted as a conversion strategy.

The first technique makes use of the `jconvertable/2` compound to explicitly specify which is the converter that should be used in a conversion operation. The first argument corresponds to the term to convert. The second argument is the term representation of a JPC converter. For example, assuming that the variable `TermToBooleanConverter` is bound to a term reifying a converter converting terms to JAVA booleans, the term `jconvertable(true, TermToBooleanConverter)` would be interpreted as the boolean `true` on the JAVA-side.

Alternatively, the `jtyped/2` compound allows to write an arbitrary term as the compound first argument and specify the expected conversion type as the second argument. In this way, the PROLOG term `jtyped(true, type(boolean))` will be interpreted as the JAVA boolean `true` instead of as the atom `'true'`.

Invoking Methods from Prolog

This section discusses how JAVA methods can be invoked from the PROLOG-side. Some examples in the rest of this chapter refer to the `Fixture` class shown in code snippet 7.19.

```

1 public class Fixture {
2
3     public static String x;
4     public static Boolean y;
5
6     public static long whatIsTheAnswer() {
7         return 42;
8     }
9
10    public static void sayHello(String name) {
11        System.out.println("Hello " + name);
12    }
13    ...
14 }
```

Snippet 7.19: A class declaring static variables of different types.

In order to invoke the `whatIsTheAnswer` method from the PROLOG-side, a programmer would need to write:

```
java::invoke(class('Fixture'), whatIsTheAnswer, term(Answer)).
```

After execution, the `Answer` variable will be bound to 42. The `term` return specifier indicates that we need in the PROLOG-side just the default conversion to a term of the returned object.

In order to invoke the `sayHello` method from the PROLOG-side, a programmer would need to write: `java::invoke(class('Fixture'), sayHello(world)).`

In this case, the method to invoke is expressed as a compound term since it includes the method arguments. There is no return specifier since the method

is of type `void`.

Accessing and Setting Fields from Prolog

This section illustrates how to make use of the methods provided by the `java/0` LOGTALK object to set and get the fields of JAVA objects.

In order to set the `x` static variable in the `Fixture` class shown in code snippet 7.19, we would need to execute in PROLOG: `java::set_field(class('Fixture'), x, true)`.

After execution, the `x` variable will be set to the JAVA String `"true"`.

If we would like to set the variable `y` that is a boolean, we would need to execute: `java::set_field(class('Fixture'), y, true)`.

Note that the two previous instructions are similar despite referring to static variables having different types. This is because the conversion of the term representing the value to set (the atom `true`) is guided by the actual type of the field on the JAVA-side.

This is an example of how we can profit from the additional information provided by a statically-typed language to infer, behind the curtains, the appropriate conversion from a PROLOG term to a JAVA object.

In order to obtain the value of the field `x`, a programmer would need to execute: `java::get_field(class('Fixture'), x, term(Answer))`.

Note that a return specifier is required, as it was the case when invoking a JAVA method from PROLOG.

Unifying Prolog Variables in Java

As discussed in chapter 2, some JAVA-PROLOG interoperability libraries allow PROLOG variables to be unified on the JAVA-side. JPC also allows this.

Consider the method `unify` in code snippet 7.20 (lines 3–6). The method receives three terms as parameters. It then unifies the first parameter with the second (line 4) and the second with the third (line 5). If the terms are not unifiable an exception will be thrown. A discussion on exception management is reserved for chapter 8.

```

1 public class Fixture {
2     ...
3     public static void unify(Term term1, Term term2, Term term3) {
4         term1.unify(term2);
5         term2.unify(term3);
6     }
7 }

```

Snippet 7.20: Unifying PROLOG variables in JAVA.

On the PROLOG-side, the method could be invoked with:

```
java::invoke(class('Fixture'), unify(term(x), term(V), term(W)))
```

First note that every parameter is wrapped in a `term/1` compound. This is to specify that the arguments should not be translated following the default

JPC conversions, but that the term representation of these arguments on the JAVA-side should be employed. If we would have written instead: `java::invoke(class('Fixture'), unify(x, V, W))` then the first atom argument would have been translated as a JAVA `String` and the PROLOG variables as JAVA `nulls`.

After the method returns, the variables `V` and `W` are unified on the PROLOG-side to the atom `x`.

7.3 Chapter Summary

In this chapter we reviewed the mechanisms offered by JPC for accomplishing a bidirectional JAVA–PROLOG integration. This library relies on the inter-language conversion constructs introduced in chapter 6.

Although JPC offers different levels of abstractions (*e.g.*, different query mechanisms), most of the integration tasks still require some boilerplate code to be written. Furthermore, the integration concern is explicit and tangled with the main functional concern of the application.

In the next chapter we review an integration framework implemented on top of JPC that reduces, and in certain cases completely eliminates, the need for explicit integration code in hybrid JAVA–PROLOG applications, thus nearing the final goal of this dissertation.

8 A Bidirectional Linguistic Integration Framework

The summit of art lies in concealing the mechanism
and the effort under the calmness of harmony.
—Auguste Antoine Bournonville, *My theatre life*

Genius is one percent inspiration,
ninety-nine percent perspiration.
—Thomas Alba Edison

In this chapter we present LOGICOBJECTS, a bidirectional linguistic integration framework built on top of JPC. The goal of LOGICOBJECTS is to improve on the explicit integration techniques provided by JPC by making the integration concern transparent and (semi-)automatic to a programmer.

In chapter 4 we discussed that conceptually an oblivious integration can be accomplished by means of inter-language conversions and a behavioural mapping of the routines in the two languages (*i.e.*, behavioural integration). JPC provides both an infrastructure for inter-language conversions (chapter 6) and a simple API for invoking routines in the foreign language and interpreting their outcome as native artefacts (chapter 7). In practice, LOGICOBJECTS just adds an extra abstraction layer making use of JPC in a transparent and (semi-)automatic way.

In this chapter we describe the integration features of LOGICOBJECTS both from the PROLOG and the JAVA perspective. The discussions of these features will follow the structure of section 4.5, where we analysed the different dimensions of behavioural integration at a conceptual level. We also included, however, some features of our framework that are related to our concrete implementation and that were not discussed before in our conceptual analysis.

8.1 From Prolog to Java with LogicObjects

The PROLOG-side of the LOGICOBJECTS framework relies on the LOGTALK delegation mechanism. This LOGTALK feature allows to ‘intercept’ any message that cannot be understood by a LOGTALK object, allowing us to transparently

delegate its execution to the JAVA-side. This is enabled per any LOGTALK object importing the `jobject` category, as illustrated in code snippet 8.1.

```

1 :- object(an_object,
2     imports(jobject)).
3     ...
4 :- end_object.
```

Snippet 8.1: A symbiotic LOGTALK object.

A high-level overview of this delegation process is illustrated in figure 8.1. In the figure, a LOGTALK object `an_object` receives a message `logtalk_message` that it does not understand. Since the object imports the `jobject` category, this message, adapted to the JAVA world, is delegated to a certain JAVA object derived from the receiver on the logic side. The details of this process are described in the rest of this section.

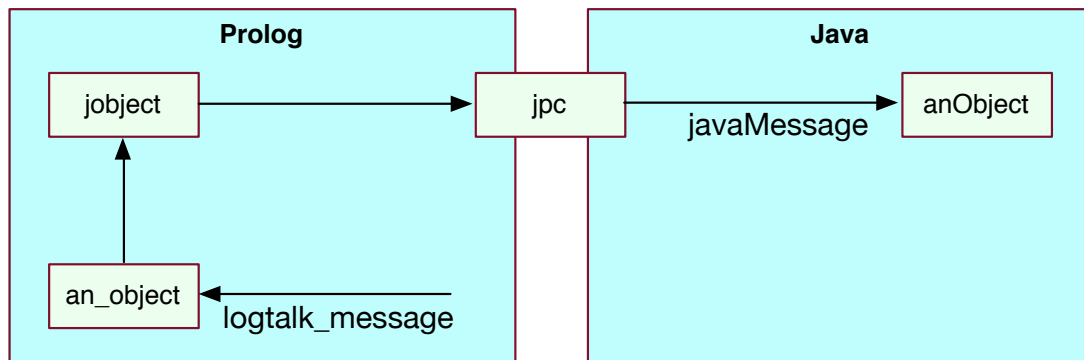


Figure 8.1: Transparent delegation of a LOGTALK message to a JAVA object.

8.1.1 Inferring the Receiver of the Method

By default, the receiver of the method on the JAVA-side is inferred automatically from the LOGTALK object receiving the message on the logic side. This term-to-object conversion is accomplished according to the procedure discussed in chapter 6, using to the default JPC conversion context described in section 7.2.1.

The programmer can customise the term to be converted to a JAVA object representing the receiver of the method. This is done by overriding the `jsself` /1 method provided by the `jobject`/0 category. If overridden, this LOGTALK method unifies its first argument to the LOGTALK object receiving the message to be delegated to the JAVA-side.

8.1.2 Inferring the Name of the Method

The method name on the JAVA-side is the same as the LOGTALK method name on the LOGTALK-side. If adapting the method name is required, the programmer needs to write the delegation code, which is trivial, as shown in code snippet

8.2. In this example, the `adaptee_method` delegates to `adapted_method`, which is not understood by the LOGTALK object and hence delegated to the JAVA-side.

```

1 :- object(an_object,
2     imports(jobject)).
3
4     :- public(adaptee_method/Arity).
5     adaptee_method(...) :- self::adapted_method(...).
6
7 :- end_object.

```

Snippet 8.2: Adapting a LOGTALK method.

8.1.3 Transforming Predicate Parameters to Method Parameters

In the same way the receiver of a LOGTALK method is automatically translated to a JAVA object, the LOGTALK method parameters are automatically translated to the parameters of the JAVA method. As discussed in section 8.1.5, a compound parameter with functor `return/1` is the only exception to this rule since it will be ignored. In case there is no matching method in the class of the receiver, a JAVA exception will be thrown.

If the parameters should be adapted before being delegated to the JAVA-side, the programmer needs to write the adaptation code as shown in code snippet 8.2.

8.1.4 Interpreting Unbound Logic Variables

By default, an unbound variable in a method parameter is interpreted as a JAVA `null`. However, any logic term, including variables, can be interpreted in JAVA as an instance of the classes reifying logic terms discussed in section 6.2.3, as long as these terms are wrapped into a `term/1` compound. For example, the following illustrates the invocation of the `unify` static method declared in code snippet 7.20.

```
class('Fixture')::unify(term(x), term(V), term(W))
```

In this example, `x` is interpreted as an instance of the `Atom` class and `V` and `W` as instances of the `Var` class reifying a PROLOG variable in JAVA.

8.1.5 Interpreting the Return Value of a Method

A JAVA method may return a value that should be captured on the logic side. There are multiple available mechanisms for a programmer to accomplish this, as discussed below.

The Return Value as a Parameter of the Logtalk Method

The parameter of a LOGTALK method with functor `return/1` is considered as the term representation of the JAVA method return value. Its only argument is

a return specifier, whose possible values have the semantics described in section 7.2.1.

For example, the message in LOGTALK: `class('MapBrowserStage')::launch(return(weak(jref(Map))))` will be interpreted as the invocation of the `launch()` static method, with no parameters, on the JAVA-side. The value returned by the method is the term representation of a JAVA weak reference, which is unified to the PROLOG variable `Map`.

The Return Value as a Parameter of the Logtalk Object

The `object/1` object can be employed to send a message to the result of evaluating a JAVA expression represented as a PROLOG term. In the simplest case, this expression consists of just a term representing an object (more complex expressions are discussed in section 8.1.7). For example, the LOGTALK message `object(class('MapBrowserStage'))::launch` is equivalent to `class('MapBrowserStage')::launch`. Although the second alternative is easier to write, it demands the explicit definition of the LOGTALK object receiving the message (in this example, `class/1` is a pre-defined object), where it should import the `object/0` category, as shown in code snippet 8.1. This is not required if we wrap the object into a `object/1` object.

The `object/2` object has a similar purpose than `object/1`, where its first argument corresponds to the term representation of a JAVA expression. The second argument of this parametric object is a return specifier which captures the return value of the method as described in section 7.2.1. For example, if the programmer wants to capture the return value of the `launch` method shown in a previous example, she could write: `object(class('MapBrowserStage'))::launch, weak(jref(Map))`, where `weak(jref(Map))` is the return specifier of the method.

The return Infix Operator

The last mechanism for capturing the return value of a JAVA method is inspired on the TUPROLOG library [44, 110] and relies on a `return` operator.

With this technique, the programmer can choose to keep both the receiver and the method parameters free of a return specifier. Instead, the return value can be captured with the `return/2` predicate, where `return` is defined as an infix operator.

The following example illustrates its usage.

```
class('MapBrowserStage')::launch return weak(jref(Map))
```

As in previous examples, `weak(jref(Map))` is the return specifier of the method `launch`.

This technique can also be applied if the receiver is a `object/1` wrapping a JAVA expression, as in the following example:

```
object(class('MapBrowserStage'))::launch return weak(jref(Map))
)
```

8.1.6 Error Handling

An exception occurring in JAVA is translated to a PROLOG error term as in any other JAVA to PROLOG conversion. If no converter has been defined for a particular exception, it will be translated to an ISO error term with the form `error(<FormalDescription>, <Context>)`.

The formal description of this error term is a compound with functor `jexception/3`, where the first argument is the term representation of the exception class, the second argument the exception message and the third the cause of the exception (represented as another exception).

The context of the error is the term presentation of the exception stack trace, which is an array of instances of the `java.lang.StackTraceElement` class. Each stacktrace element is in turn reified as a compound with functor `st/3`, where the arguments correspond to the class where the exception was thrown, the method name and the line number where the exception was originated.

8.1.7 Java Expressions in Prolog

In this section we describe the available JAVA expressions that can be written in PROLOG. These expressions can be written in any of the expression place holders described before (*e.g.*, the first argument of a `jobject/1-2` object or the first argument of the `eval/1-2` or `invoke/2-3` methods of the `java/0` object).

Constructor Expressions

As in XPCE [138], a constructor call can be written as a compound `class_name(<constructor arguments>)`. For example, `'java.lang.String'(hello)` represents a call to the constructor of a JAVA `String` passing as an argument the string `"hello"`.

A limitation is that this expression type cannot be employed for representing a constructor with empty arguments. This is because in standard PROLOG implementations compound terms cannot have zero arguments. Atoms should not be employed to denote a zero arguments constructor since it would be no way to distinguish between a constructor with zero arguments and a legitimate string having as its value the name of a JAVA class.

As an alternative, a programmer can write an expression denoting the invocation of a constructor with zero arguments as: `class(class_name)::new` or `type(class_name)::newInstance`.

Method Call Expressions

A method call is represented with the `::` LOGTALK operator. For example, the expression `an_object::a_method(a::b)` represents the invocation of the method `a_method` parameterised with the result of invoking the method `b` on the object `a`. In fact, the argument of a method can be an arbitrarily complex expression, not being limited to a simple object or another method call.

Cascading Expressions

LOGTALK provides a construct for message cascading (*i.e.*, sending several messages to the same object). Although in JAVA there is no such a construct ¹, an expression can consist of the LOGTALK construct representing such a cascade and will be evaluated with the expected semantics on the JAVA-side.

The following is an example of a message cascading:

```
java::eval(x::(a,b))
```

Here, the object representation of the term *x* will receive on the JAVA-side first the message *a* and then the message *b*. If the return value of a cascading expression is required (*e.g.*, it is the argument of a method call) its return value is considered to be the result of evaluating the last message sent to the receiver.

Note that this is functionally equivalent to the shorter expression:

```
x::(a,b)
```

However, there is an important difference performance-wise. While the first expression requires only one round-trip to the JAVA world, the second call relies on the LOGTALK message cascading mechanism, therefore resulting in two separate calls to JAVA. In addition, the second expression requires that the object *x* is defined on the logic side and imports the *jobject/0* category.

Broadcasting Expressions

LOGTALK also provides a construct for message broadcasting (*i.e.*, sending the same message to several objects).

The following is an example of a message broadcasting:

```
java::eval((x,y)::a)
```

Or:

```
jobject((x,y))::a
```

Here, the object representations of first the term *x* and then the term *y* will receive on the JAVA-side the message *a*. If the return value of a broadcasting expression is required (*e.g.*, it is the argument of a method call) its return value is considered to be the result of evaluating the message sent to the last receiver.

Note that this is functionally equivalent to the shorter expression:

```
(x,y)::a
```

However, as in the previous case, there is an important difference performance-wise. While the first expressions require only one round-trip to the JAVA world, the last alternative relies on the LOGTALK message broadcasting mechanism, therefore resulting in two separate calls to JAVA.

Sequence Expressions

Sequence expressions allow to evaluate a comma separated sequence of expressions. For example, we could write:

```
java::eval((x::a, y::b))
```

¹Some other object-oriented languages like SMALLTALK do have a kind of message cascading construct.

This will first evaluate the expression `x::a` and then `y::b`. If the return value of a sequence expression is required (*e.g.*, it is the argument of a method call) its return value is considered to be the result of evaluating the last expression in the sequence.

Note that this is functionally equivalent to the shorter:

`x::a, y::b`

As in previous cases, this shorter alternative have an impact on performance. It requires two separate calls to the JAVA-side. In addition, it requires that the objects `x` and `y` are defined in the logic side and import the `jobject/0` category.

Accessors and Mutators Expressions

Accessor and mutator expressions provide a short notation for accessing and setting the fields of objects.

The following expression represents the value of the field named `a` in the object `x`:

`x::[a]`

If the field name `a` is a number and the receiver `x` an array or object implementing the `java.util.List` interface, the expression represents the element of the array or list at the given index.

The following expression sets the value of the field named `a` in the object `x` to the value `v`:

`x::[a,v]`

If the field name `a` is a number and the receiver `x` an array or object implementing the `java.util.List` interface, the expression sets the value of the element of the array or list at the given index. If the return value of a mutator expression is required (*e.g.*, it is the argument of a method call) its return value is considered to be the receiver `x` of the expression.

Note that while a mutator expression can play the role of a goal since it has a side effect, an attempt to write an accessor expression as a goal will result in an error.

8.1.8 Explicit Reflection

In the previous sections we have reviewed constructs and expressions that allow a programmer to easily execute JAVA expressions from PROLOG. Particularly, the object `jobject/1-2` allows us to send messages to the object resulting from evaluating an arbitrarily complex JAVA expression.

One of the limitations of this approach is that no JAVA method having the name of a symbol forbidden as a LOGTALK message can be invoked (*e.g.*, high-order PROLOG predicates such as `findall/3`). An alternative, is to build a term representing a message call expression and make use of the `eval/1-2` LOGTALK method on the `java\0` object. For example, instead of writing: `x::m` we can always write: `java::eval(x::m)`.

As an alternative notation, we provide the object `robject/1-2` which makes explicit certain reflective operations on an object (*e.g.*, invoking a method, or setting and getting fields).

This object provides more explicit field manipulation methods in comparison to the shorter (but more cryptic) expressions relying on `[]/1-2`.

Invoking Methods

The `invoke/1` LOGTALK method allows to explicitly invoke a method in JAVA. An example of its usage is: `robject(x)::invoke(m(...))`

If the return value of the method is required, the alternative variant:

`robject(x, ReturnSpecifier)::invoke(m(...))` captures such return value according to a certain return specifier.

Getting Fields

The `get_field/1` LOGTALK method allows to get the field of an object. The only argument corresponds to the field name or index (if the object is an array or a list). The return value is set according to the return specifier given in the second argument of the receiver `robject/2` object.

The following goal gets the value of the field named `a` in the object `x` according to a `ReturnSpecifier`:

```
robject(x, ReturnSpecifier)::get_field(a)
```

If the field name `a` is a number and the receiver `x` an array or object implementing the `java.util.List` interface, the field value corresponds to the element of the array or list at the given index.

Setting Fields

The `set_field/2` LOGTALK method allows to set the field of an object. The first argument corresponds to the field name or index (if the object is an array or a list) and the second the field value.

The following goal sets the value of the field named `a` in the object `x` to the value `v`:

```
robject(x)::set_field(a,v)
```

If the field name `a` is a number and the receiver `x` an array or object implementing the `java.util.List` interface, this method sets the value of the element of the array or list at the given index.

8.1.9 Choosing the Right Constructs

In this section we have provided several mechanisms for interacting with a JAVA program from the logic perspective. Some of them were redundant. This is because it has proven complex to find a single integration approach that is best suited for all possible scenarios. This urges for a taxonomy of techniques that guide a programmer to the best integration technique for a specific problem.

Scenario 1: Evaluating non Method-Call Expressions

Description: An expression that is not written as a LOGTALK message call needs to be evaluated.

Solution: The only alternative is to use the `eval/1-2` LOGTALK method in the `java\0` object.

Example: The evaluation of the sequence expression: `java::eval((x,y))`.

Scenario 2 : Bypassing the Logtalk Sending Mechanism

Description: An expression needs to bypass the LOGTALK message sending mechanism.

Solution: The only alternative is to use the `eval/1-2` LOGTALK method in the `java\0` object.

Example: The evaluation of a message cascading expression so it will be evaluated in one single trip to JAVA: `java::eval(x::(a,b))`.

Scenario 3 : Sending Messages to Expressions

Description: A message needs to be sent to an object which is written as an expression.

Solution: The `jobject/1-2` object provides this functionality. It can wrap an arbitrarily complex expression and it will delegate any message to the object resulting of the evaluation of such expression.

Example: Sending a message to a sequence expression: `jobject((x,y))::a`.

Scenario 4 : Sending Messages with Names Forbidden in Logtalk

Description: A message with a name forbidden by LOGTALK needs to be sent to an object.

Solution: The method `invoke/1` in the `robject/1-2` object provides this functionality. It can wrap an arbitrarily complex expression and the argument of `invoke/1` can be any term that can be interpreted as a method call in JAVA.

Example: Sending the `findall/3` message to an arbitrary object: `robject(x)::invoke(findall(p1,p2,p3))`

Scenario 5 : Sending Messages with Non Forbidden Names

Description: A message with a non-forbidden name needs to be sent to an object.

Solution: Import the `jobject/0` category in the object and just send the message as any other LOGTALK message. Make use of the `return` operator if the method return value should be captured.

Example: Assuming that `x` imports the `jobject/0` category, `a` is an undefined method in `x` and `ReturnSpecifier` is a return specifier: `x::a return ReturnSpecifier`.

Scenario 6 : Get and Set Fields in Expressions

Description: The field of an object needs to be accessed or mutated as part of an expression.

Solution: Use accessor and mutator expressions by means of the `[]/1-2` operator.

Example: Getting the field value `a` in the object `x`, where the value of `a` is the argument of a method call: `y::m(x::[a])`.

Setting the field value `a` to `v` in the object `x`, where `x` is the argument of a method call: `y::m(x::[a,v])`.

Scenario 7 : Get and Set Fields in Goals

Description: The field of an object needs to be accessed or mutated as a goal.

Solution: Use the explicit `get_field/1` and `set_field/2` methods in the `robjct/1-2` object.

Example: For getting the field value `a` in the object `x` with the return specifier `ReturnSpecifier: robjct(x, ReturnSpecifier)::get_field(a)`.

For setting the field value `a` to `v` in the object `x`: `robjct(x)::set_field(a,v)`.

In the next section we review the integration from the JAVA language perspective.

8.2 From Java to Prolog with LogicObjects

The JAVA-side of the LOGICOBJECTS framework is based on the use of annotations [91]. They provide a general mapping between JAVA objects and PROLOG terms and between JAVA methods and PROLOG predicates.

We will develop this section taking as a starting point the *London Underground* case study described in section 3.1.

8.2.1 Inferring the Receiver on the Logic Side

In section 3.1 we provided the implementation of a `line/1` LOGTALK object (code snippet 3.4, page 57) representing a line in a subway system. In this section we introduce, using LOGICOBJECTS, the JAVA counterpart of this LOGTALK object. The `Line` class is shown in code snippet 8.3. On line 1, the annotation `LObject` identifies this class as (partially) implemented in logic (*i.e.*, a symbiotic class). The `LObject` annotation includes an optional `name` attribute indicating the name of the LOGTALK object on the logic side implementing the abstract methods of this JAVA class. When provided, this name will be the one used. Otherwise, its name will be the one of the JAVA class.

When the object on the logic side is a parametric object, its parameters need to be declared on the JAVA-side by means of an `args` attribute in the `LObject` annotation. In the `Line` class example, this attribute is present in the `@LObject` annotation. It maps the instance variable `name` to the single parameter of the

parametric object `line` on the logic side. An instance of the JAVA class `Line` with its instance variable `name` set to “*central*” is thus automatically translated to the logic term `line(central)`. In this example, the transformation of the object property `name` to a term is straightforward, as it is just a string.

If the property would have been a symbiotic object itself (*e.g.*, in case the field’s declaration includes a `LObject` annotation) this transformation process will continue recursively, given that the property object could also have properties that are symbiotic objects and so on.

```

1 @LObject(name = "line", args = {"name"})
2 public abstract class Line {
3     private String name;
4
5     public Line(String name) {
6         this.name = name;
7     }
8
9     @LMethod
10    public abstract boolean connects(Station s1, Station s2);
11
12    @LMethod(name = "connects", args = {"_", "_"})
13    public abstract int segments();
14 }

```

Snippet 8.3: The `Line` object in JAVA.

Translating a term to an object (*e.g.*, when interpreting the result of a query as the return value of a method) is the inverse process. If a term name equals the name of a symbiotic class and its arity corresponds to its number of arguments (according to the `LObject` annotation), it is converted to an instance of such class. If a public constructor with the same number of arguments as the term arity exists, it is reflectively invoked passing as arguments the term arguments converted to objects. If there is not a matching constructor, the class will be instantiated using the default (no args) constructor. In this case, the instance properties having the names described in the `args` attribute of the `LObject` annotation are set according to the term arguments.

8.2.2 Inferring the Predicate Name

In LOGICOBJECTS, methods are mapped by default to logic predicates with the same name and arity. An example of this mapping is found in the `connects(Station, Station)` method (code snippet 8.3, lines 9–10). Since this JAVA method has two parameters, it is mapped to the LOGTALK method `connects/2` in code snippet 3.4 (lines 14–16, page 57). When invoked, this LOGTALK method will be executed in the context of the LOGTALK object corresponding to the JAVA object receiving the message (*i.e.*, a `line/1` LOGTALK object).

However, a programmer can always customise this mapping by means of the arguments of an `@LMethod` annotation. The JAVA method `segments()` illustrates this (code snippet 8.3, lines 12–13). The `name` attribute of this annotation indicates the name of the logic predicate and the number of elements in the `args` attribute represents the predicate arity. Therefore, in this example

the `segments()` method will also be mapped to the logic predicate `connects/2`.

With this technique, we are thus able to map a single LOGTALK predicate, `connects/2`, to different JAVA methods: `segments()` and `connects(Station, Station)`, according to our needs. The semantics of these mappings is explained in section 8.2.6.

8.2.3 Transforming Method Parameters to Predicate Parameters

The parameters of a JAVA symbiotic method are automatically translated to terms as specified in section 8.2.1. For example, the stations which are the parameters of the `connects(Station, Station)` method (code snippet 8.3, line 10) are automatically translated to terms of the form `station(station_name)`.

Alternatively, a programmer can have a more fine-grained control over which are the parameters of the logic predicate by means of the `args` argument of the `@LMethod` annotation. An example is shown for the `segments()` method (line 12), where the arguments of the `args` attribute in the `@LMethod` annotation are specified as unbound PROLOG variables, which take us to the next integration dimension.

8.2.4 Passing Unbound Variables as Predicate Arguments

In PROLOG, it is common to write queries with unbound variables. In JAVA, however, all variables must be bound to a value (either an explicitly assigned value, or a default initialisation value). Consider the `segments()` method mentioned before. Its arguments are explicitly specified by means of the `args` attribute of the `@LMethod` annotation. These arguments are interpreted as PROLOG terms. In this case, both parameters are the symbol “_”, which is interpreted as an anonymous logic variable on the logic side.

The class `Station` (code snippet 8.4) shows examples of methods having as arguments non-anonymous variables. For instance, the predicate to which the method `connected` (lines 9–10) is mapped, takes as first argument a logic variable *LSolution* and as second argument the first parameter received by the JAVA method.

To refer to that first parameter of the JAVA method we use the macro expression `$1`. In general, the symbol `$n` (where $n \in \mathbb{N}$) is interpreted as the object received as n^{th} parameter by the JAVA method. The macro expressions currently available in our framework are listed in section 8.2.10.

In the next section we will see how we can use variables to define the return value of a symbiotic method.

```

1 @LObject(args = {"name"})
2 public abstract class Station {
3     private String name;
4
5     public Station(String name) {
6         this.name = name;
7     }

```

```

8
9     @LMethod(args = {"LSolution", "$1"})
10    public abstract Station connected(Line line);
11
12    @LComposition @LMethod(args = {"LSolution"})
13    public abstract List<Station> nearby();
14
15    @LMethod(name = "reachable", args = {"$1", "LSolution"})
16    public abstract List<Station> intermediateStations(Station station);
17 }

```

Snippet 8.4: The `Station` object in JAVA.

8.2.5 Interpreting a Query Solution as an Object

As discussed in section 4.5.2, the result of a logic query can be seen as a set of frames binding logic variables to terms, where each frame corresponds to one logic solution. The solution of a symbiotic method is a transformation from this set of frames to a JAVA object. By default, a JAVA object representation of the first logic solution (the first frame) is considered by LOGICOBJECTS as the symbiotic method's return value. In this section we discuss different techniques for instantiating such JAVA object from a single logic solution. The composition of a set of solutions is discussed in section 8.2.6.

Inferring Return Values from a Logic Variable Name

Our first heuristic is based on a naming convention: if one of the logic variables in a query has as name *LSolution*, its binding in the frame of the first solution will be considered as the term representation of the JAVA object to return.

As an example, reconsider the implementation of the method `connected(Line)` in the `Station` class (code snippet 8.4, lines 9–10, 158). This method is mapped to the LOGTALK method `connected/2` (code snippet 3.5, line 20). As specified by the `args` attribute of the `@LMethod` annotation, the predicate's first argument is a PROLOG variable *LSolution* and the second argument is the term representation of the first parameter of the JAVA method. Upon evaluation of the query, the *LSolution* variable will be bound to a compound term of the form `station(name_station)`. Given the convention introduced above, the return value of the symbiotic method will be the transformation of this term to a JAVA object according to the conversion discussed in section 8.2.1.

Inferring Return Values from Method Signatures

If no variable with name *LSolution* is found in the query, the framework will attempt to infer its return value from its signature. The term representation of this value has as name the method name and as arguments the parameters of the method. The implementation of the `Metro` class illustrates this. The `line` method (code snippet 8.5, line 7) is mapped to a method with the same name on the logic side. In case the LOGTALK method succeeds, the framework will consider as the solution to the method the logic term `line` having as argument

the only string parameter of the method. This term will be converted to an instance of the `Line` class as discussed in section 8.2.1. In case a line with the name given as a parameter of the JAVA method does not exist in the logic world, the method will throw a `NoSuchElementException` exception.

```

1 public abstract class Metro {
2     //returns a list with all lines
3     @LComposition @LMethod(name="line", args={"L"})
4     public abstract List<Line> lines();
5
6     //returns an existing line with a given name
7     public abstract Line line(String s);
8 }

```

Snippet 8.5: The `Metro` class in JAVA using LOGICOBJECTS.

Explicit Specification of Return Values

The previous heuristics reduce the amount of explicit mappings that need to be specified by a programmer. However, we do provide an `@LSolution` annotation to let a programmer specify explicitly the term representation of the JAVA object to return, overriding the heuristics presented above. This term can be of arbitrary complexity and refer to as many logic variables as required.

For instance, if we had wanted to encode explicitly the heuristics for returning the logic variable *LSolution* as the return value of the `connected(Line)` method (code snippet 8.4, lines 7–8), we could have done so by annotating it with `@LSolution("LSolution")`. Since this is the default mapping for the solution, it can be omitted, but if an alternative or more complex solution is desired, this can be defined explicitly with the `@LSolution` annotation as well.

8.2.6 Non-Determinism Support

The previous section illustrated how LOGICOBJECTS infers the return value of a symbiotic method from the first solution of a logic routine. This subsection discusses how to compose a value from multiple solutions, or from properties of the logic solution set.

It is not trivial to infer that a method should return a composition of multiple solutions (*e.g.*, as a list) instead of a single solution. In the first stage of our research, we tried to infer this from the method return type. For example, if the method returns a collection class, then the intention of the programmer *may be* that the method returns a collection of results rather than a single result. This assumption is not always valid, however.

Consider, for example, the method `intermediateStations(Station)` in the `Station` class (code snippet 8.4, lines 15–16). This method is mapped to the predicate `reachable/2` in the `station` LOGTALK object (code snippet 3.5, line 38). The `args` attribute in the `@LMethod` annotation indicates that the first argument of the LOGTALK method will be the logic term representation of the first parameter of the JAVA method (indicated by the macro-expression `$1`). The second argument is the PROLOG variable *LSolution*. As explained in

section 3.1.1, upon execution of the LOGTALK method, the *LSolution* variable is bound to a list with the intermediate stations between the receiver station object and the station object passed as first argument. The method return value is the value bound to that variable in the first solution, according to the heuristics discussed in section 8.2.5. The JAVA method thus returns a list of objects that corresponds to the binding of one variable in *one* solution (the first) answered by the LOGTALK query. This is an example where a method returning a collection of objects is not intended to answer a single collection of different solutions, but rather a single solution consisting of a collection of objects.

In order to resolve ambiguities between both ways of interpreting collections, LOGICOBJECTS provides the `@LComposition` annotation. The JAVA method `nearby()` (code snippet 8.4, lines 12–13) in class `Station` is an example of the usage of this annotation (line 12). This method is mapped to the LOGTALK method `nearby/1` which takes as argument an unbound logic variable *LSolution*. On the logic side, the unbound variable passed as argument will be bound to a station nearby the receiver station object. On the JAVA-side, as in the previous example, a binding of the *LSolution* variable corresponds to the term representation of an individual solution. Given the `@LComposition` annotation, the framework considers the type of the method (a `List` class) as a container of all its solutions.

Another example is the `lines()` method in the `Metro` class (code snippet 8.5, lines 3–4). In this case, the arguments of the method do not include an *LSolution* variable, neither an `@LSolution` annotation. Therefore, the term representation of each solution is given by the name and arguments of the LOGTALK method (given explicitly by the `name` and `args` attributes of the `@LMethod` annotation). As in the previous example, the `@LComposition` annotation will instruct the framework to collect all these individual results in a collection. In both cases, the framework will choose a collection class implementing the JAVA `List` interface, given that this is the return type of the method.

Finally, the return value of a method could be inferred from properties of the complete logic solution set. For example, in case when none of the heuristics discussed in this section can be applied, the framework will inspect the return type of the method. If this is a numeric type, the return value will be the number of results of the query (*e.g.*, the `segments()` method in class `Line`). If it is a boolean, the method answers whether the query produces at least one solution (*e.g.*, the `connects(Station, Station)` method in class `Line`).

8.2.7 Error Handling

Behind the curtains, all queries generated by LOGICOBJECTS are wrapped in an ISO PROLOG `catch/3` predicate. In case of an error on the PROLOG-side, the error term representation is automatically converted to a JAVA exception and thrown on the JAVA-side.

Figure 8.2 illustrates some automatic mappings between common ISO PRO-

LOG errors and JAVA exceptions. As the figure shows, a PROLOG error with the form `error(domain_error(Type, Term), Context)` will be interpreted as a JAVA `DomainError` exception. Any non-recognised ISO error (*i.e.*, an error with the form `error(FormalDescription, Context)`) will be interpreted as a JAVA `UnknownIsoPrologError` exception. Finally, any non-ISO error will be interpreted as a `PrologError` exception.

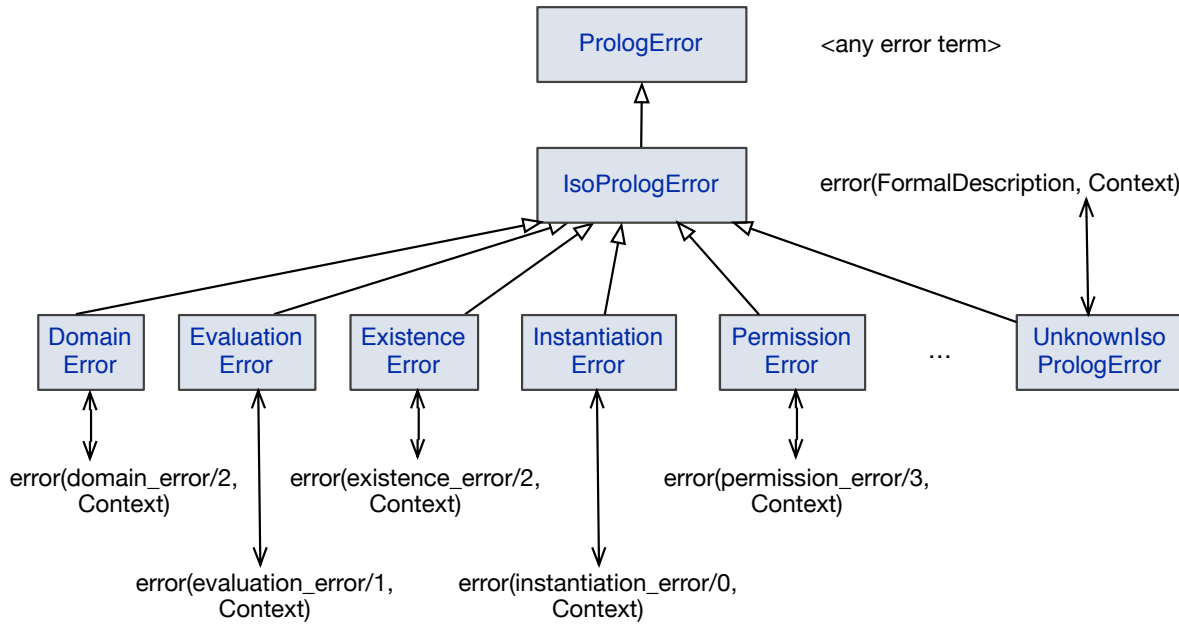


Figure 8.2: Automatic mapping between JAVA exceptions and PROLOG errors.

8.2.8 Instantiating Symbiotic Classes

To use our framework, a programmer simply needs to instantiate symbiotic classes using a provided factory method. Everything else, including the transparent import of dependencies on the logic side, is automatically managed using runtime code generation and byte code instrumentation techniques [30].

As an example, code snippet 8.6 shows an instantiation of a logic class and the invocation of a logic method. The first argument of the factory method corresponds to the logic class to instantiate. The other arguments correspond to the logic class constructor parameters, if any. In this code snippet, the output corresponds to the number of segments on the line *central*, as specified on the logic side (code snippet 3.1).

```

1 import static org.logicobjects.LogicObjects.newLogicObject;
2 ...
3 Line line = newLogicObject(Line.class, "central");
4 System.out.println("Number of segments: " + line.segments());

```

Snippet 8.6: Instantiating a symbiotic class.

How exactly this is done will not be discussed in detail in order not to make this document lengthier than it already is.

8.2.9 Auto-loading Logtalk objects

Usually, in PROLOG we need to load a file containing rules and facts before we are able to query and reason about them. Likewise, in Logtalk, objects are usually defined in their own file. Normally, there is a loader file that loads all the application objects, as exemplified in code snippet 3.6 (page 58).

Our framework provides several mechanisms for transparently loading LOGTALK objects before they can be used. When instantiating a symbiotic class, if the framework detects that a LOGTALK file with the same name exists in the same package as the class, it will transparently load it. This happens only the first time when the class is instantiated. If the LOGTALK files are in a different directory, or if they must be loaded in a specific order, the loader can make use of a `logicobjects.properties` configuration file. This file should be located in the same package as the symbiotic classes. In our example, this configuration file contains just the line:

```
imports=logic_lib/example/metro/load_all
```

It is possible to include additional files by separating them with a comma. It is also possible to specify a LOGTALK library as a compound term. For instance, the following line indicates that the LOGTALK `types_loader` library is required:

```
imports=library(types_loader)
```

If none of these techniques is enough for defining the LOGTALK dependencies of a symbiotic class, the programmer could use the `imports` attribute in the `LObject` annotation described in section 8.2.1. This attribute defines the LOGTALK files that should be loaded before the symbiotic class can be instantiated.

8.2.10 Macros

Term expressions are typically used when adapting method parameters or when defining a method's return value. Previous examples (listing 8.4) have shown that these expressions often need to include the representation as terms of certain JAVA objects. To facilitate this task, the following macros can be used:

- \$\$** A comma separated list of the original arguments of the JAVA method, converted into terms.
- \$0** The object receiver ('this') of the method.
- \$n** The n^{th} parameter of the method as term.

8.2.11 Integration with plain Prolog

Our framework also offers an `LQuery` annotation for symbiotic interactions with logic libraries. This annotation uses a `value` attribute that can include any arbitrary PROLOG query.

As an example, consider a method annotated with:

```
LQuery("predicate1($0), predicate2($1)")
```

This implies that the annotated method will be interpreted, in the logic side, as a query of the logical conjunction of the predicate `predicate1` having as argument the receiver of the JAVA method, and the predicate `predicate2` having as argument the first parameter of the JAVA method. All the other annotations (*e.g.*, `LSolution`, `LComposition`) are, of course, still available and with the same semantics.

8.2.12 Choosing the Right Constructs

In this subsection we describe a guide for choosing the best mechanism for interacting with a PROLOG program from the JAVA perspective.

The two first criteria concerns the way JAVA artefacts should be reified into PROLOG. The last one is based on performance requirements.

Scenario 1: Reification Based on the Mapping of Object Properties to Term Arguments

Description: It is possible to establish a mapping between instances of a symbiotic class and a term, such that the properties of those instances correspond to arguments in the term.

Solution: Annotate the code with the LOGICOBJECTS annotations introduced in this chapter and instantiate the class with the factory method described in section 8.2.8.

Scenario 2: Alternative Reification Strategies

Description: There is not a correspondence between object properties and term arguments in the mapping between instances of a symbiotic class and a term.

Solution: Employ the JPC high-level API described in section 7.1.1 to accomplish the integration.

Scenario 3: Integration of Performance Critical Code

Description: The classes to integrate are part of the performance critical code of the application.

Solution: Employ the JPC low-level API described in section 7.1.1 to accomplish the integration. This API does not rely on automatic inter-language conversions and make the integration code completely explicit. Since no inferences are required in the integration, this API is more efficient than the higher-level layers of JPC and LOGICOBJECTS.

Note that this improvement on performance affects only the call to foreign routines (*e.g.*, logic queries) and the interpretation of their results. Neither

LOGICOBJECTS nor the high-level JPC API impose any penalty on the execution of the foreign routine itself. A discussion on performance follows in section 9.1.

8.3 Discussion

The framework introduced in this chapter provides a semi-automatic bidirectional integration mechanism between PROLOG and JAVA. It is not fully automatic, since in certain cases the programmer needs to explicitly customise the integration. A custom integration can be declaratively specified on both sides. In JAVA, by means of annotations. In PROLOG, by importing a LOGTALK category and by overriding or implementing certain predicates.

A remaining problem concerns the automatic adaptation of names (*e.g.*, method names) respecting the conventions of the two languages. In an earlier implementation we set an automatic adaptation of names based on the expected naming conventions. Therefore, a JAVA method name `myMethod` written using camel case would be translated to PROLOG as `my_method`, where tokens in the name are separated by underscores.

Unfortunately, this did not prove to be a perfect solution. For example, consider a JAVA method name `getURL`. This may be translatable as the LOGTALK method name `get_url` or `get_u_r_l`. The first one is more readable, but its default translation back to a JAVA method name is `getUrl`, which is incorrect. The second one guarantees a correct translation, but it is difficult to write and read.

Therefore, in the current version we have suppressed such automatic translation of names. We will allow a programmer to specify, per artefact, a name conversion strategy as part of our future work.

8.4 Chapter Summary

In this chapter we finally discussed our high-level bidirectional integration framework for PROLOG and JAVA. This framework is built on top of our lower-level integration library providing services for mapping inter-language artefacts and executing foreign routines.

In the next chapter we complete the thesis by validating the different aspects of our integration approach, before wrapping up in chapter 10.

9 Validation

All life is an experiment.
The more experiments you make the better.
—Ralph Waldo Emerson

In this chapter we describe a validation of the tools and techniques discussed in this dissertation. The validation of each integration perspective (*i.e.*, the PROLOG and JAVA perspective) was accomplished by the implementation of the case studies discussed in chapter 3. We collected some metrics on the reduction in programming effort to illustrate better the simplification brought by our approach.

Although our main focus has been to provide high-level programming abstractions, we also collected some performance metrics to get some idea of the potential performance penalties introduced by the architecture and techniques of our tools.

We also validated our portability claims by means of another case study requiring the interaction with multiple instances of distinct PROLOG engines at the same time.

9.1 Validation of the Integration from the Object-Oriented Side

Using the LOGICOBJECTS library, we re-implemented our *London Underground* case study originally implemented with the JPL library in section 3.1. This case study modelled the underground system of the city of London, and allows a user to query different relations between underground lines and stations.

As discussed in section 3.1.2, our original JPL-based implementation required a significant amount of boilerplate code to write a JAVA class with methods implemented in logic. In contrast, table 9.1 shows a notable reduction in code size, and thus in programming effort, that can be gained by using our LOGICOBJECTS framework.

Note that the difference in number of lines of code for classes is due not only to the amount of code in the different methods that implement invocations to logic routines, but also partially stems from the need for additional auxiliary

methods in JPL that are responsible for mapping objects, such as lines 4–12 in code snippet 3.7, as well as explicitly loading PROLOG files. Our framework instead generates all this infrastructure transparently when required.

The table 9.1 also compares the result of a stress test accomplished in both implementations. We show the difference in execution time required by each pair of corresponding methods in the two implementations.¹ the differences in processing time can serve as an approximate measure of the *adaptation effort* (*i.e.*, a measure of the complexity of adaptation heuristics in different scenarios). However, this difference should be interpreted with care. Concretely, LOGICOBJECTS, at the time of writing, does not provide a native driver optimised for its requirements. Instead, it relies on existing libraries (*e.g.*, the JPL-based driver) for interacting with concrete PROLOG engines.

There are many factors that influence the difference in the adaptation effort in the distinct methods. For example, methods that do not require an adaptation of their parameters (*i.e.*, not including an `args` attribute in a `@LObject` annotation) are the ones with less impact on execution time (*e.g.*, the `connects` method in class `Line` and the `line` method in class `Metro`). On the other hand, methods using macro expressions are among the ones with greater increase in execution time (*e.g.*, the `connected` and `intermediateStations` methods in class `Station`). In addition, the adaptation effort is greater in methods manipulating collection of objects (*e.g.*, the `connected` method in class `Station` and the `lines` method in class `Metro`), since it grows proportional to the amount of objects (also requiring adaptation) in such collections. A discussion of these observations follows in section 9.5.

¹Tests accomplished with a 2.8 GHz Intel Core 2 Duo processor and 4 GB of RAM.

Class	Line	Station		connected		nearby		stations		Metro	
Method		connects	segments								
#LOC in JPL	30	7	7	64	14	14	14	19	40	14	10
#LOC in LOGICOBJECTS	10	1	2	14	2	2	2	2	9	2	1
LOGICOBJECTS annotations (*)	LO	LM	LM	LO	LS, LM	LC, LS, LM	LC, LM	LS, LM	LO	LC, LM	LM
time JPL (**)		1.9	1.7		1.9	2.5		1.7		30.0	1.9
time LOGICOBJECTS (**)		12.9	13.6		43.8	38.4		44.3		38.4	11.6
$\Delta t \approx$ adaptation effort		11.0	11.9		41.9	35.9		42.6		35.4	9.7

Table 9.1: A comparison between LOGICOBJECTS and JPL in the *London Underground* case study.

(*) LO = LObject, LM = LMethod, LS = LSolution, LC = LComposition).

(** time in seconds, 50000 executions).

9.2 Validation of the Integration from the Logic Side

In this section, we show how a PROLOG programmer can interact with JAVA artefacts by means of LOGICOBJECTS. We will use the MAPQUERY application [24] discussed in section 3.2 as our case study for showing the PROLOG-side features of LOGICOBJECTS in action. This application allows a programmer to query geographical data (*e.g.*, *nodes* and *ways*) imported from the OSM project [69].

This case study introduces an interesting variation with respect to the one discussed in the previous section. We will assume we cannot modify the original source code of the classes conforming our model on the JAVA-side, which is a common scenario when reusing existing components. Since we cannot annotate those classes, we will show how a custom conversion context can be configured so that the required conversions occur transparently.

9.2.1 Configuring the Default Conversion Context

For this application we need to define a default conversion context that can translate between representations of the main artefacts in our problem. This conversion context is configured with bidirectional converters for coordinates (code snippet 9.1), nodes (code snippet 9.2) and ways (code snippet 9.3).

The `CoordinatesConverter` converts between instances of the `Coordinates` class and a compound with functor `coordinates/2`. The conversion of a term to a `Coordinates` instance (lines 5–10) consists of interpreting the arguments of the term as double values (*i.e.*, a longitude and latitude) and using them for instantiating the `Coordinates` class. Since the default conversion of a `FloatTerm` (which is the class of the compound arguments) is a JAVA `double`, there is no need to specify the required target type of the conversion. Also note that no casting is required to assign the resulting value to a JAVA object or, in this case, a primitive type.

The conversion of a `Coordinates` instance to a term (lines 12–15) consists on creating a compound term with name `coordinates` having as arguments two integer terms corresponding to the longitude and latitude of the `Coordinates` instance.

```

1 public class CoordinatesConverter implements ToTermConverter<Coordinates,↵
    Compound>, FromTermConverter<Compound, Coordinates> {
2
3     public static final String COORDINATE_FUNCTOR_NAME = "coordinates";
4
5     @Override
6     public Coordinates fromTerm(Compound term, Type type, Jpc jpc) {
7         double lon = jpc.fromTerm(term.arg(1));
8         double lat = jpc.fromTerm(term.arg(2));
9         return new Coordinates(lon, lat);
10    }
11
12    @Override

```

```

13     public Compound toTerm(Coordinates coordinates, Class<Compound> termClass, Jpc jpc) {
14         return jpc.toCompound(COORDINATE_FUNCTOR_NAME, asList(coordinates.getLon(), coordinates.getLat()));
15     }
16 }

```

Snippet 9.1: The Coordinates converter.

The Node and Way converters are implemented in a similar way. The `NodeConverter` converts between instances of `Node` and a compound with functor `node/3`. The `WayConverter` converts between instances of `Way` and a compound with functor `way/3`.

```

1 public class NodeConverter implements ToTermConverter<Node, Compound>, FromTermConverter<Compound, Node> {
2
3     public static final String NODE_FUNCTOR_NAME = "node";
4
5     @Override
6     public Node fromTerm(Compound term, Type type, Jpc jpc) {
7         long id = jpc.fromTerm(term.arg(1));
8         Coordinates coordinates = jpc.fromTerm(term.arg(2));
9         Map<String, String> tags = jpc.fromTerm(term.arg(3), Map.class);
10        return new Node(id, coordinates, tags);
11    }
12
13    @Override
14    public Compound toTerm(Node node, Class<Compound> termClass, Jpc jpc) {
15        return jpc.toCompound(NODE_FUNCTOR_NAME, asList(node.getId(), node.getCoordinate(), node.getTags()));
16    }
17 }

```

Snippet 9.2: The Node converter.

```

1 public class WayConverter implements ToTermConverter<Way, Compound>, FromTermConverter<Compound, Way> {
2
3     public static final String WAY_FUNCTOR_NAME = "way";
4
5     @Override
6     public Way fromTerm(Compound term, Type type, Jpc jpc) {
7         long id = jpc.fromTerm(term.arg(1));
8         List<Long> nodesIds = jpc.fromTerm(term.arg(2));
9         Map<String, String> tags = jpc.fromTerm(term.arg(3), Map.class);
10        return new Way(id, nodesIds, tags);
11    }
12
13    @Override
14    public Compound toTerm(Way way, Class<Compound> termClass, Jpc jpc) {
15        return jpc.toCompound(WAY_FUNCTOR_NAME, asList(way.getId(), way.getNodesIds(), way.getTags()));
16    }
17 }

```

Snippet 9.3: The Way converter.

Once the converters have been defined, they should be registered into a conversion context and such context should be configured as the default conversion context introduced in section 7.2.1. An example of this is shown in code snippet

9.4.

```

1 Jpc context = JpcBuilder.create()
2   .register(new CoordinatesConverter(), new Functor(↵
    CoordinatesConverter.COORDINATE_FUNCTOR_NAME, 2).asTerm())
3   .register(new NodeConverter(), new Functor(NodeConverter.↵
    NODE_FUNCTOR_NAME, 3).asTerm())
4   .register(new WayConverter(), new Functor(WayConverter.↵
    WAY_FUNCTOR_NAME, 3).asTerm())
5   .build();
6 Jpc.setDefault(context);

```

Snippet 9.4: Configuring the MapQuery conversion context.

9.2.2 The city/1 Logtalk Object

Some examples in this section refer to the auxiliary `city/1` LOGTALK object shown in code snippet 9.5. This object provides a method for obtaining the coordinates (*i.e.*, a `coordinates/2` LOGTALK object) of a city. In this code snippet, we can see that coordinates with latitude 4.3524950 and longitude 50.8467493 correspond to the centre of the city of Brussels.

```

1 :- object(city(_Name)).
2
3     :- public(coordinates/1).
4     coordinates(coordinates(4.3524950, 50.8467493)) :-
5         parameter(1, 'brussels').
6     ...
7 :- end_object.

```

Snippet 9.5: The `city/1` LOGTALK object.

9.2.3 Loading Geographical Data from an OSM File

On the PROLOG-side, the first step for interacting with the MAPQUERY application consists in loading some geographical data from a file adhering to the OSM format. An `OsmDataLoader` utility class providing a `load` method for loading such an OSM file into a PROLOG engine was introduced in code snippet 3.9 (lines 9–11). To be instantiated, this loader class requires a PROLOG engine in its constructor.

```

1 prolog_engines::this_engine(Engine),
2 jobject(class('org.jpc.examples.osm.OsmDataLoader')::new(Engine))::load(↵
    file('/Users/sergioc/Documents/workspaces/heal/mapquery/src/main/↵
    resources/org/jpc/examples/osm/brussels_center_filtered.osm')).

```

Snippet 9.6: Loading OSM data into a PROLOG engine.

A reference to a JAVA object reifying the current PROLOG engine can be obtained by means of the `this_engine/1` method in the `prolog_engines/0` LOGTALK object. After passing this object to the constructor of the `OsmDataLoader` class, the `load` method can be invoked on the resulting object passing as a parameter the file to load.

Code snippet 9.6 shows these steps. On line 2, the `load` method is invoked on the result of evaluating the expression `class('org.jpc.examples.osm.OsmDataLoader')::new(Engine)`, which is interpreted as a new instance of the `OsmDataLoader` class. Alternatively, this expression could have been replaced by `'org.jpc.examples.osm.OsmDataLoader'(Engine)`, which denotes a constructor expression as discussed in section 8.1.7. The argument of the `load` method is a term that will be interpreted as an instance of `java.io.File`. It is not required to write the longer constructor expression `'java.io.File'('.../brussels_center_filtered.osm')` since, `File` being a common JAVA type, a simpler compound having as functor `file/1` can be employed instead.

9.2.4 Opening the MapQuery Browser

A MAPQUERY browser can be opened by invoking the static method `launch` on the class `org.jpc.examples.osm.ui.MapBrowserStage`. Code snippet 9.7 shows the invocation of this method. Alternatively, the programmer could have written:

`class('...MapBrowserStage')::launch(return(weak(jref(Map))))`, with the same semantics.

Note that, in order for this browser to be useful, we need a reference to it so we can control it from the PROLOG-side. Therefore, the invocation of the method also includes the return specifier `weak(jref(Map))`. As discussed in section 7.2.1, this return specifier indicates that the term bound to the `Map` variable is a black box reference to a JAVA object (*i.e.*, the MAPQUERY browser window). Furthermore, this term can be dereferenced as long as the object is not garbage-collected on the JAVA-side (*i.e.*, the term points to a JAVA weak reference).

```
1 class('org.jpc.examples.osm.ui.MapBrowserStage')::launch return weak(jref←
    (Map)).
```

Snippet 9.7: Opening the MAPQUERY browser.

9.2.5 Zooming and Moving the Map

Before querying the map for points of interest, we need to zoom it to a convenient level and move it to the area which is of our interest. An example of this is illustrated in code snippet 9.8, where we assume that the `Map` variable is unified with a term that is interpreted as a reference to the MAPQUERY browser (*i.e.*, an instance of the `MapBrowser` class introduced in section 3.2.1).

The method `zoomTo` is invoked sending the integer 14 as an argument (line 1). Afterwards, we query the coordinates of the city of Brussels and unify them with the variable `BRU` (line 2). Finally, we move the map to the Brussels coordinates (line 3).

```
1 Map::zoomTo(int(14)),
2 city(brussels)::coordinates(BRU),
```

3 Map::goTo(BRU)

Snippet 9.8: Zooming and Moving the Map.

9.2.6 Querying and Drawing in the Map

Now we would like to query certain points of interest and show them on the map. For example, we would like to know all the ways (*e.g.*, streets) that are located close to the *Bruxelles-Central* railway station. This query is illustrated in code snippet 9.9. First we obtain one node (line 1) which is identified as a railway station (line 2) and which name in french corresponds to '*Bruxelles-Central*' (line 3). We also obtain the coordinates of such node (line 4). Afterwards, we query one way (line 5) which is located near those coordinates with a maximum distance of 100 meters (line 6). Finally, we invoke the map `draw` method, which expects as arguments on the JAVA-side a list of `Taggable` objects (*i.e.*, nodes or ways). The conversion of the terms bound to the `Node` and `Way` PROLOG variables occur automatically since the default conversion context has been configured to translate these terms to instances of the `Taggable` class.

```

1 osm::node(Node),
2 Node::has_tags([railway-station]),
3 Node::has_tags(['name:fr'-'Bruxelles-Central']),
4 Node::coordinates(Coordinates),
5 osm::way(Way),
6 Way::near(Coordinates, 0.1),
7 Map::draw([Node, Way]).

```

Snippet 9.9: Querying and Drawing the Map.

9.2.7 Wrapping up

Figure 9.1 illustrates a screenshot of an SWI PROLOG console showing the results of executing the queries discussed in this section. The figure shows 4 queries, labelled with the numbers 2 to 5. (the query #1 loading the drivers on the PROLOG-side is not shown).

The query #2 loads OSM data into the current PROLOG engine (code snippet 9.6). Query #3 opens a MAPQUERY browser (code snippet 9.7). Query #4 zooms and moves the map (code snippet 9.8). This and the next query make use of a convenient feature of SWI where a variable obtained in a previous query can be referenced in subsequent queries by prefixing its name with the symbol `$`. Finally, query #5 gathers and draws the points of interest described in section 9.2.6 (code snippet 9.9). In the example, each result is drawn at a time. Only the first result of the last query is shown here, but there are about 10 solutions being drawn on the map.

After obtaining the *first* result of the last query, the map looks as shown on figure 9.2, with only one node and way in the centre highlighted in orange. After all the results of the query have been gathered, the interface will look as was shown on figure 3.5.

```

2 ?- prolog_engines::this_engine(Engine),
|   jobject('org.jpc.examples.osm.OsmDataLoader'(Engine))::load(file('/Users/sergioc/Documents/
workspaces/heal/mapquery/src/main/resources/org/jpc/examples/osm/brussels_center_filtered.osm')).
% /Users/sergioc/.jpc/prolog-engine-writer168286270869289730.tmp compiled 2.59 sec, 41,831 clauses
Engine = jref_term(0)
false.

3 ?- class('org.jpc.examples.osm.ui.MapBrowserStage')::launch return weak(jref(Map)).
Map = jref_term(1)
false.

4 ?- $Map::zoomTo(int(14)), city(brussels)::coordinates(BRU), $Map::goTo(BRU).
BRU = coordinates(4.352495, 50.8467493)
false.

5 ?- osm::node(Node), Node::has_tags([railway-station]), Node::has_tags(['name:fr'-'Bruxelles-
Central']), Node::coordinates(Coordinates), osm::way(Way), Way::near(Coordinates, 0.1),
$Map::draw([Node, Way]).
Node = node(29751692, coordinates(4.3570678, 50.84548), ['addr:country'-'BE', name-'Bruxelles-Central
- Brussel-Centraal', 'name:de'-'Br  ssel Hauptbahnhof', 'name:en'-'Brussels Central Station',
'name:fr'-'Bruxelles-Central', 'name:nl'-'Brussel-Centraal', railway-station, ... - ...|...]),
Coordinates = coordinates(4.3570678, 50.84548),
Way = way(144502506, [1580594539, 1580594544, 1580594538, 1580594525, 1580594535, 1580594539], [])

```

Figure 9.1: Interacting with the MAPQUERY interface from a PROLOG terminal.

In this section we explored the LOGICOBJECTS integration with a focus on the logic programming perspective. In the next section we show a last case study that we implemented to validate the portability of our approach.

9.3 Portability Validation

As a validation of the portability of our library, we implemented HYDRA [19], an open-source multi-engine PROLOG query browser. HYDRA allows a programmer to interact with many instances of different kinds of PROLOG engines.

The HYDRA implementation is not strongly coupled with these engines, to the point that new PROLOG engine drivers can be added at runtime. Instead, it is implemented on top of the JPC abstraction of a PROLOG engine presented in section 5.3. HYDRA can be (re-)used either as an out-of-the-box tool or embedded into another application.

9.3.1 Hydra Components

HYDRA provides an interface to query different PROLOG engines concurrently. The components of this interface are shown in figure 9.3. We describe them below.

9.3.2 Configuring the Prolog Engine Creation

The HYDRA settings pane (figure 9.3, #1) allows to configure the creation of a PROLOG engine. For example, it has options for specifying a PROLOG (or LOGTALK) file that should be loaded every time a new PROLOG session is started.

In addition, it allows to specify if LOGTALK should be automatically pre-loaded in these PROLOG sessions. We will come back to LOGTALK related features in section 9.3.9.

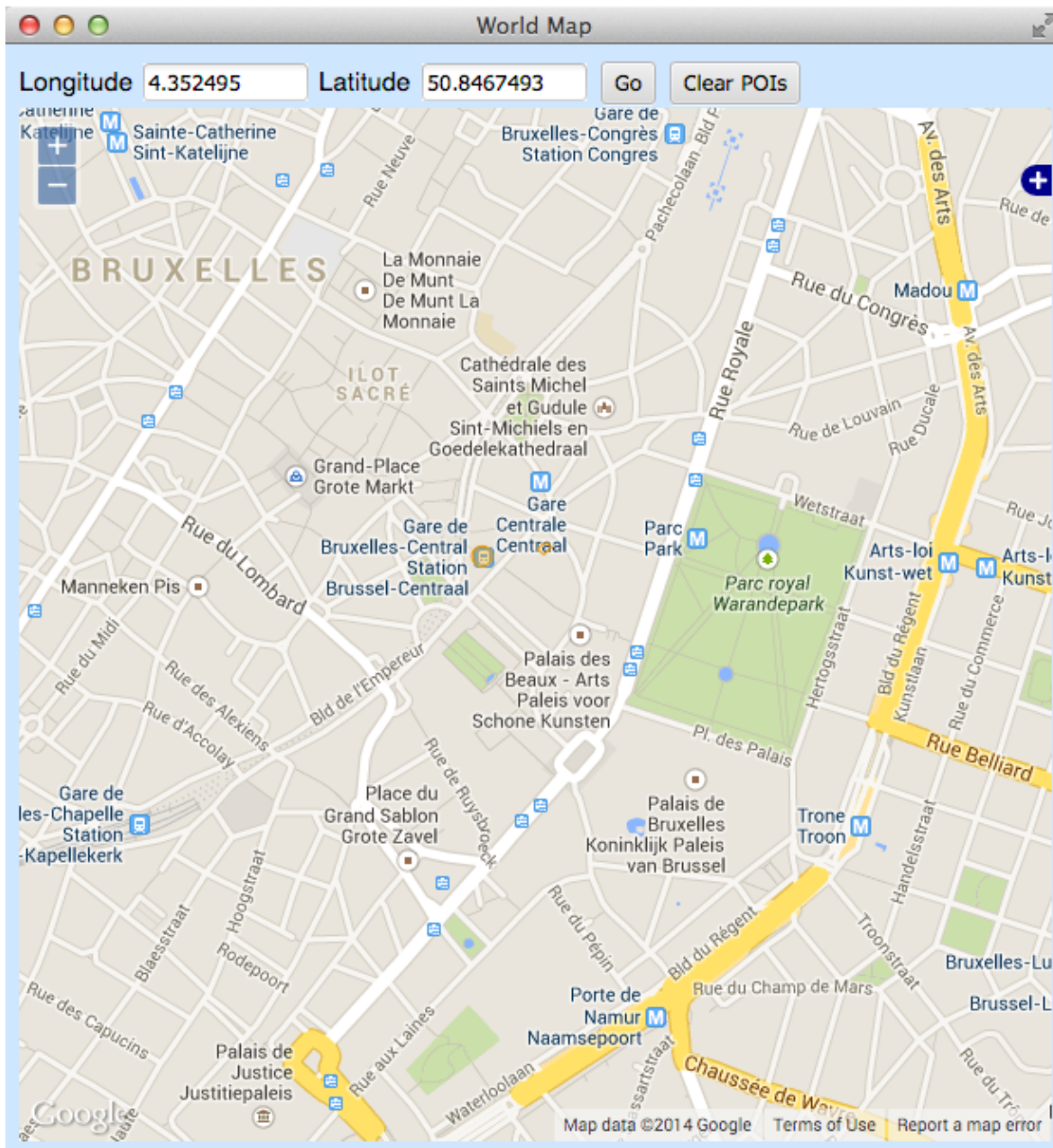


Figure 9.2: Querying nodes and ways with the MAPQUERY interface.

9.3.3 Managing Drivers

The drivers browser (figure 9.3, #2) allows to inspect the currently loaded drivers. In the current implementation, a list pane on the left allows to select a PROLOG engine provider (*e.g.*, SWI, YAP or XSB). The available drivers, for the selected PROLOG engine provider, are shown in the list pane to the right. In the figure they are referred as INTERPROLOG², JPL and PDT CONNECTOR, since these were the bridge libraries employed to build these drivers.

New drivers can always be added at runtime with the button labeled #4. A file chooser will then be opened and a user can select multiple jar files encapsulating JPC drivers to load in Hydra.

²Although we built drivers on top of the three INTERPROLOG ports for SWI, YAP and XSB, only the public version of INTERPROLOG for XSB works correctly at the time of writing.

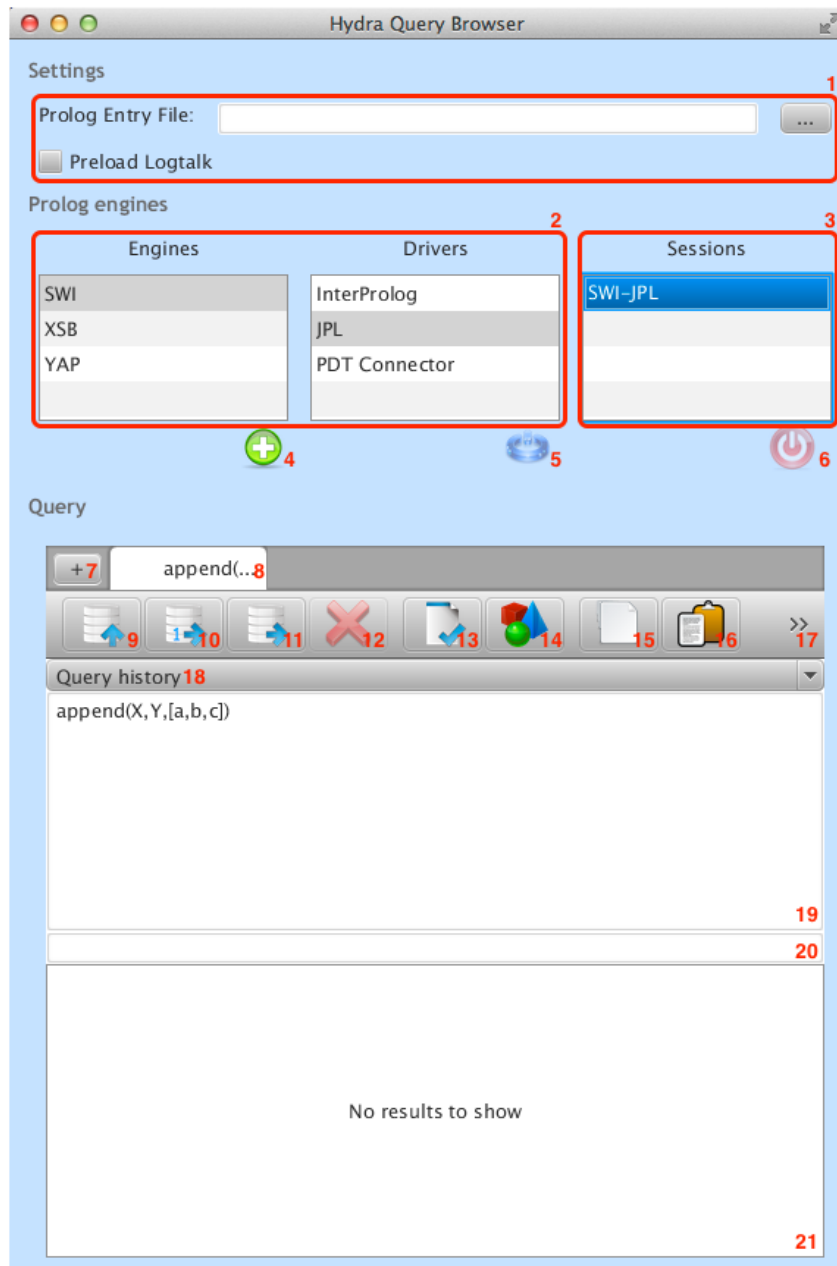


Figure 9.3: The HYDRA multi-engine query browser.

9.3.4 Managing Prolog Sessions

Once a driver has been selected, a new PROLOG session can be started with the button labeled #5. In figure 9.3, a PROLOG session has been started in SWI PROLOG by means of a driver based on the JPL library. Note that the button for creating new sessions is disabled since JPL does not support the creation of more than one PROLOG session per JVM. However, other drivers may support it (*e.g.*, the InterPROLOG or PDT Connector based drivers).

A session can be closed with the button labeled #6. In this case any resource associated with it should be freed. Such a session can be closed only if its driver allows it (which is not the case for the selected JPL based driver, but it is possible for the other available ones).

9.3.5 Management of Prolog Queries

PROLOG queries can be spawned in the bottom half of the HYDRA interface. Multiple tabs (for example the one labeled #8) represent, each of them, a PROLOG query. New query definitions can be created with the button labeled #7. The text of the query is written in the text area labeled #19. A history of all previously executed queries in the current PROLOG engine is available in the combo box labeled #18.

In the previous figure, the query has not been executed yet. Figure 9.4 shows what the interface looks like after pushing the *next solution* button (figure 9.3, #11). Once a solution has been found (or if no solution was available), the status is updated accordingly in the text area labeled #20. For example, figure 9.5 shows the status after all the query solutions have been exhausted.

Note that after a next solution has been requested, a new query cannot be started (in the current tab) until the current query solutions have been exhausted or the query was canceled with the button labeled #12. However, a concurrent query can be started at any moment in a different tab for the same session if the driver employed for communication with the PROLOG engine has multi-threading support.

The cancel button mentioned above, in addition to resetting the state of a query, allows to interrupt an ongoing query that may have taken too much time to finish. Again, this is only possible if the driver of the PROLOG engine where the query is executed allows for interruption of queries.

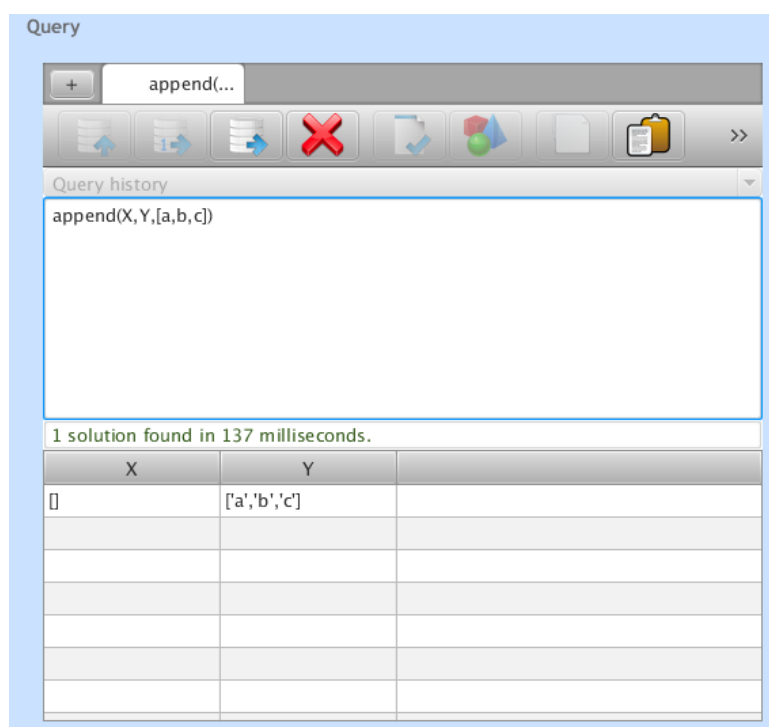


Figure 9.4: Querying for the next solution.

In addition to providing options for inspecting the next solution of a query, HYDRA has convenient buttons for presenting just one solution (figure 9.3, #10) or all solutions (figure 9.3, #9). The *one solution* button is functionally

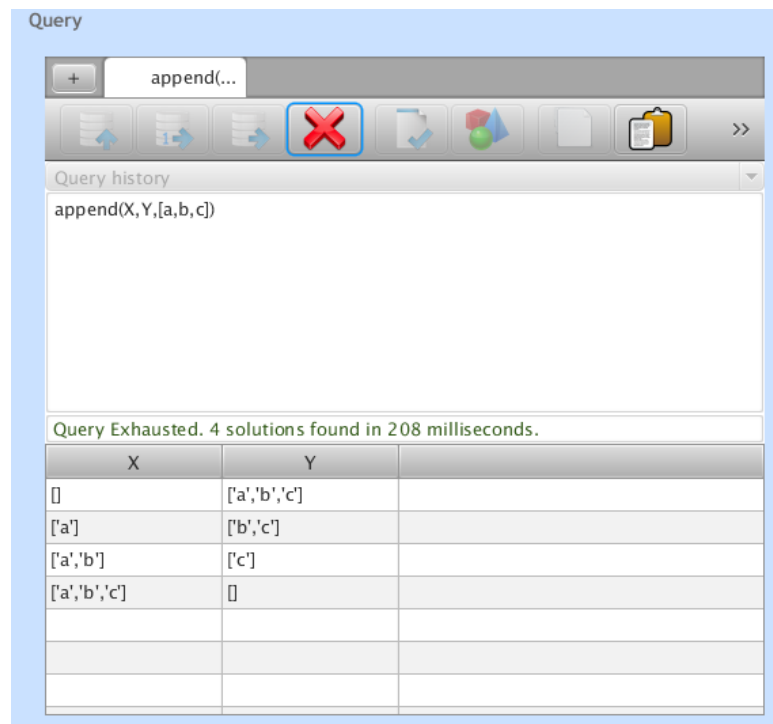


Figure 9.5: Exhausted solutions.

equivalent to pushing once the next solution button and resetting the query afterwards. However, depending on the implementation of the driver, it may be more efficient to request one solution than requesting the next solution and then resetting the query. The *all solutions* button is in general functionally equivalent to pushing the next solution button until the query is exhausted.

9.3.6 Query Edition Options

Basic edition support is provided by the query browser, such as cleaning the query text area (button #15) or copying the current query to the clipboard (button #16). Some buttons in the toolbar are hidden (see the arrow labeled #17). They concern additional editing options such as saving the current query to a file or opening a previously saved query.

9.3.7 Presenting Query Results

Query solutions are presented in the bottom of the query pane (figure 9.3, #21). In the example shown in figures 9.4 and 9.5, these solutions are presented as a table having as columns all the non-anonymous PROLOG variables present in the query. Rows represent a solution to the query, where each cell binds a term to a PROLOG variable.

If the programmer wants to filter the variables to be shown in the query, he just has to replace the name of non interesting variables with the anonymous variable (represented by the underscore character). For example, figure 9.6 shows all the solutions of the same query shown in figure 9.5, but only the variable named *Y* is shown, since the variable *X* has been renamed to *_*.

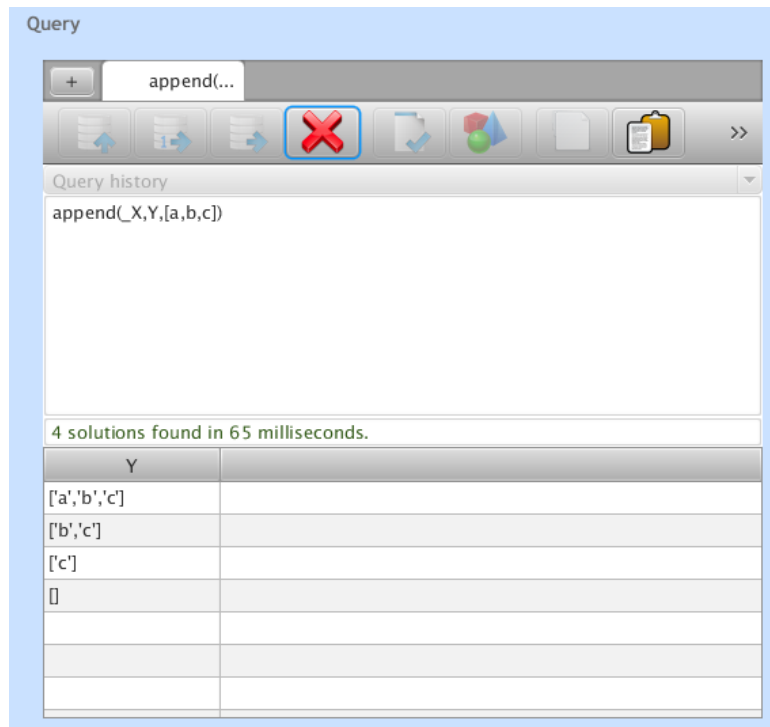


Figure 9.6: All solutions with an anonymous variable.

9.3.8 Loading Files

Although commands for loading files can be directly written and executed in the query text area, this is a repetitive and error prone activity, since it requires writing the path of files in the filesystem. What is worse, there are semantic differences between the implementations of common loading predicates such as `consult/1`.

In order to facilitate this task, HYDRA supports the generation of source file loading queries. Its interface provides a button labeled #13 in figure 9.3 that opens a file chooser allowing to select one or more source files to load. Furthermore, the chosen files can be a mixture of PROLOG and LOGTALK files, and HYDRA will generate the appropriate loading commands according to the user selection.

9.3.9 Logtalk Support

Attempting to provide support for PROLOG modules would have been an overly complex task given the different ways modules are implemented, when implemented, across different PROLOG engines.

Instead, HYDRA provides support for interacting with LOGTALK. This satisfies our portability goals regarding a module system since LOGTALK provides a highly portable object-oriented layer that is compatible across several PROLOG implementations.

The button labeled #14 will open a LOGTALK library browser (figure 9.7). It provides an (alphabetically ordered) view of all the currently available LOGTALK libraries and allows to select some of them to be loaded in the current

PROLOG session. If some libraries are selected in the browser, a `logtalk_load/1` predicate call will be generated having as argument a list with such libraries.

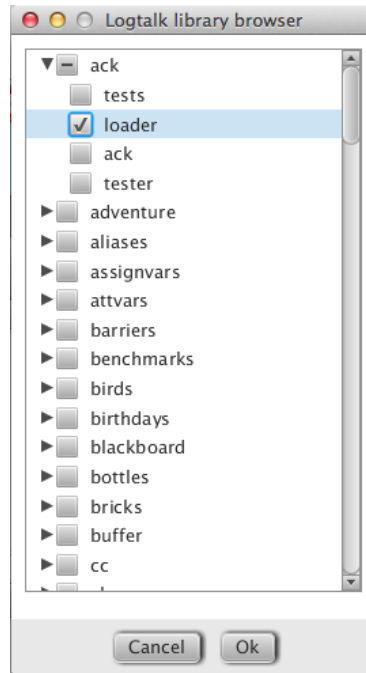


Figure 9.7: The LOGTALK library browser.

Our LOGTALK library browser is dynamic: if the available LOGTALK libraries change at runtime, it will reflect this change the next time it is opened.

Until this point we have presented three case studies validating different aspects of both the JPC and LOGICOBJECTS library. In the next section we discuss the threats to validity we found during the development and evaluation of this work.

9.4 Threats to Validity

In this section we discuss the open questions and weaknesses of our research that were found during its development and validation. A further analysis of some of these threats to validity follow in the next section.

9.4.1 Limits regarding the Transparency and Automation of the Integration

LOGICOBJECTS allows, in many cases, to infer the best integration approach and automatically execute and interpret the results of foreign routines in PROLOG and JAVA. This is possible thanks to the implementation of inter-language conversion inference heuristics that take into consideration many factors, such as the internal structure of a term to convert, or the type of the object to which a term should be converted. In addition, it provides a considerable catalog of default converters that minimises the work of a programmer when executing operations that involve common types.

However, a completely automatic and transparent integration is only possible under the assumption that there is only one correct way to map inter-language artefacts which is, unfortunately, not the case. As we have seen, sometimes the explicit intervention of the programmer is required to customise how the integration should be accomplished.

9.4.2 Limits of Type Inference

We have found that type-guided conversions are a powerful mechanism that minimises the integration effort in many cases. For example, we use this technique when determining how the result of a symbiotic method in JAVA (usually expressed as a term) should be converted to a JAVA object. In this case, the return type of the JAVA method guides this conversion.

This also applies when setting the fields of a JAVA object. In this case, the conversion of a term to the field is guided by the field type.

However, there are limits at this inference mechanism. For example, it is not a good decision to attempt to infer the conversion of a term to a JAVA method parameter based on the parameter type. This is because the method declaring such parameter could be overloaded. Therefore, overloaded methods with the same number of parameters but distinct parameter types, may exist in the method class or its super classes.

It could be argued that type-guided parameter conversions could be applied in methods that are not overloaded. But this is a fragile solution since the conversion may change if a new overloading method is added to the JAVA class, or its super classes, in the future. Therefore, we have preferred to avoid type-guided conversion inferences concerning method parameters.

9.4.3 Lack of Causal Connection in the Reification of Foreign Artefacts

Embedded approaches such as SOUL maintain a causal connection between an artefact and its representation in the foreign language. This can be accomplished by means of the definition of a special kind of term that wraps an object. The reification as a term of this object depends on the current object state. In the case of SOUL, an object does not define its term reification but its unification strategy (*i.e.*, how the object is unified with logic terms). In this way, an object term may unify with other kind of logic terms (*e.g.*, a compound term).

There are two main advantages regarding the preservation of causality in the reification: (1) the terms bound to logic variables in the result of a logic query can be trivially converted back to objects since they *are* (or wrap) the objects and (2) there is no need to synchronise the term reification of an object during any given query, since this reification is automatically kept up to date in case the object mutates.

Our approach does not attempt to maintain such a causal connection since this would require a non-portable technique (assuming the existence of a special

term wrapping an object) and a specific architecture (the logic engine and the object-oriented program sharing the same memory space).

In section 9.5.2 we discuss to what extent this is a limitation.

9.4.4 Complexity on Understanding and Debugging Type-Guided Conversions

As discussed in section 6.4, a limitation of our approach concerns some difficulty regarding the traceability of the converters, inferences and factories employed in a given conversion operation. This may hinder the debugging and maintenance of hybrid JAVA-PROLOG programs in certain scenarios. Particularly, this problem is evident when employing converters with crosscutting domains and ranges in multi-inheritance hierarchies.

9.4.5 Context-Depending Conversions Issues

Both JPC and LOGICOBJECTS are built on top of the concept of a conversion context. This context determines how inter-language conversions are applied between JAVA and PROLOG artefacts.

Although the JPC library can work with many contexts at the same time, the current version of LOGICOBJECTS employs one default context at any given time. However, this context can be changed if required.

9.4.6 Performance Issues

In spite of the reduction in program size observed in table 9.1, the increase in execution time is still considerable. This may hinder the adoption of the advanced features of our library in a production setting. Possible optimisation paths are discussed in the next section.

9.4.7 Portability Issues

An unfortunate aspect of a PROLOG-based approach are portability issues due to the lack of strong, comprehensive, official and de facto standards. These problems are minimised to a certain extent by the usage of LOGTALK as a portability layer.

A current problem regarding portability concerns the PROLOG to JAVA integration direction. In that integration direction we were not able to test our approach with XSB, due to a currently unfixed garbage-collector bug.

We will accomplish the missing tests with XSB regarding the PROLOG to JAVA integration as soon as this issue is solved.

9.5 Discussion

In this section we discuss our observations during the validation of our work.

9.5.1 Transparency and Automation

As discussed in the previous section, for scenarios demanding a highly customisable integration it was not possible to accomplish a complete oblivious integration. However, we minimise the explicit writing of inter-language conversions by providing a mean to specify mappings between inter-language artefacts in a declarative way (*e.g.*, using source-code annotations). In addition, the work of a programmer is greatly reduced thanks to the runtime generation of integration code in JAVA and the transparent delegation to a LOGTALK object accomplishing the integration in PROLOG.

9.5.2 Causal Connection vs. Inter-Language Conversions

Our approach is based on the notion of context-dependent inter-language conversions. We observed in section 9.4.3 that alternative approaches preserving a causal connection in the reification of foreign artefacts present some advantages.

One advantage was related to the automatic conversion of terms back to objects. As discussed in section 7.1.2, our approach also allows this by means of establishing mappings between terms and object references. Although the programmer needs to explicitly specify such mappings, this is not a disadvantage in comparison to techniques based causal connection. This is because in the latter technique a programmer also needs to define the term reification of an object or the object unification strategy.

The other advantage of approaches based on causal connection concerned the automatic synchronisation of the term reification of an object in case it mutates. This feature introduces as many issues as advantages. For example, a programmer should deal with non-monotonic reasoning techniques since the logic theory can mutate during the execution of a query. This also introduces problems in multi-threaded applications since a sort of synchronisation mechanism may be needed in order to define locks between queries and methods mutating the state (hence the term representation) of certain objects.

9.5.3 Complexity of our Approach

In section 6.4, we highlighted certain convoluted scenarios in which our approach becomes complex and maybe difficult to maintain.

However, in scenarios not using these advanced features, understanding how type-guided conversions are accomplished does not present such complexity. Conversely, our approach offers a very intuitive mechanism for accomplishing several kinds of inter-language conversions. A proof of this is the success in industry of other context-dependent and type-guided inter-language conversion library (Google's GSON library [61]) that inspired some of the techniques presented in this dissertation.

Complex JPC advanced features should be managed with care and remain reserved for scenarios that demand such flexibility and customisability. However, JPC offers different level of abstractions. When automatic inter-language

conversions are not appropriate, the programmer can always switch to the lower-layer API if the problem is better suited for explicit integration code.

9.5.4 Performance Optimisation Paths

There are many optimisation paths to follow in order to reach an acceptable performance in a production setting. The most evident is through the implementation of native drivers that do not rely on existing bridge libraries and that are optimised for the LOGICOBJECTS requirements.

Another possibility of improvement regarding performance is to cache mappings between artefacts so they do not have to be calculated every time.

Finally, note that our framework does not impose any overhead in the execution of a logic routine per se, which is often the real bottleneck performance-wise, but on the automatic adaptation of its arguments and the interpretation of its results as objects. In addition, this automatic adaptation occurs only when using the advanced features of LOGICOBJECTS. These features can be transparently replaced by the lower-level API provided by JPC in critical integration points demanding a high performance.

At this point we have preferred to avoid any premature optimisation and focused our efforts on the transparency and automation of the integration.

9.6 Integration Features Revisited

A description of the integration features found in our related work was depicted in table 2.1. These features were compared according to the criteria established in section 2.8.2. In this section, table 9.2 revisits these features and includes in the comparison our LOGICOBJECTS library.

As discussed in chapter 8, LOGICOBJECTS provides a significant level of transparency and automation in the integration, both from the JAVA and PROLOG perspective. This is reflected in the columns related to the *Obliviousness* dimension.

Chapter 7 discussed the support provided by LOGICOBJECTS to the integration features related to the *State Sharing* and *Behavioural Integration* dimensions.

The integration features concerning the interaction with *Multiple Environments* depend on the bridge libraries currently used to build our drivers (figure 5.4). Since some of these bridge libraries do support the interaction with multiple PROLOG engines, this feature is inherited from our drivers, and hence from JPC and LOGICOBJECTS. However, none of our drivers currently support the interaction with multiple JVMs. We may add support to this feature in the future in case we find an interesting case study where this is required.

Finally, the integration dimensions regarded as *Orthogonal Qualities* concern the portability and customisability of the surveyed approaches. The portability aspects of our approach were discussed in chapter 5. Customisability aspects

		Completeness	Obliviousness	State Sharing	Behavioural Integration	Multiple environments	Orthogonal Qualities
		OO to Logic Language (2.7.1) Logic Language to OO (2.7.1)	Obliviousness from the Logic Language(2.7.1) Obliviousness from the OO Language(2.7.1)	Sharing Unbound Logic Variables (2.7.2) Sharing Object State (2.7.2)	Invoking Methods from the Logic Language (2.7.3) Querying Predicates from the OO Language (2.7.3) Evaluating OO Expressions from the Logic Language (2.7.3) Executing Complex Queries from the OO Language (2.7.3) Non-determinism Support(2.7.3)	Multiple Logic Engines (2.7.4) Multiple OO Environments (2.7.4)	Portability (2.7.6) Customisability (2.7.5)
LOGICOBJECTS							
Non Embedded Logic Engines							
JPL							
INTERPROLOG							
PDT CONNECTOR							
JASPER							
Embedded Logic Engines							
TUPROLOG							
P@J + TUPROLOG							
JINNI							
SOUL							
Object-Oriented Libraries							
XPCE							

Table 9.2: Comparative Table of Integration Features Revisited.

were mostly presented in chapter 6 by means of the introduction of a customisable inter-language conversion context (section 6.1.6).

The table also shows that, in many aspects, our approach is a super set of most of the surveyed interoperability techniques. This guarantees, to certain extent, the completeness of our libraries.

We should highlight, however, that the table only shows features related to general language interoperability, since this has been the scope of this dissertation. Other features related to, for example, advanced inter-language reflection are supported by other integration techniques (*e.g.*, SOUL) and not by our approach. As discussed in chapter 10, some of these features are part of our future work.

9.7 Cross-Fertilisation from Several Domains

LOGICOBJECTS profited from the cross-fertilisation of ideas from several domains. Many of these ideas have already been extensively validated in industry by other libraries. For example, the success of Google’s GSON library [61] validates the inter-language conversion architectural pattern employed by our approach, since we adapted and generalised that pattern to be applicable to our domain (*i.e.*, inter-language conversions between JAVA and PROLOG artefacts).

The same applies to other architectural patterns employed by LOGICOBJECTS such as the one of *drivers*, that has been extensively used in, for example, libraries allowing object-oriented programs to communicate with databases (*e.g.*, Java–Database Connectivity (JDBC) [54]).

The usage of JAVA annotations for declaratively defining mappings between objects and other artefacts has also been employed in other widespread and successful libraries, such as JAXB [77] or HIBERNATE [4].

9.8 Chapter Summary

In this chapter we presented three case studies focused on (1) the integration from the JAVA perspective; (2) the integration from the PROLOG perspective and (3) our portability objectives.

We demonstrated our two different methodologies for specifying inter-language conversions. The first one consisted on the usage of annotations, which can be employed when a programmer can modify the source code of symbiotic classes. The second one demands the explicit configuration of a conversion context and is useful when there is no access to the source code of symbiotic classes.

We presented some relevant threats to the validity of our approach and discussed why and how our approach overcomes or alleviates them.

10 Conclusions

Everything should be made
as simple as possible,
but not simpler.
—**Albert Einstein**

In this final chapter we revisit our initial research questions and discuss how they were answered in this dissertation. Following this discussion we present future work and elaborate our conclusions.

10.1 Our Research Questions Revisited

In section 1.3 we introduced the research questions that guided the development of this work. We revisit them in this section and discuss how we provided an answer to them in this dissertation.

RQ.1 What is the degree of integration provided by existing interoperability approaches and what is the architecture of such approaches?

Our related work discussed in chapter 2 described several integration approaches between logic and object-oriented languages. We observed that, roughly, there are two main types of architecture of these integration approaches: object-oriented programs interacting either with embedded or non-embedded logic engines.

Furthermore, in table 2.1 we observed a certain correlation between the architecture of a hybrid system and its integration level. This correlation was not surprising since systems integrating logic engines embedded into an object-oriented language profit from the considerable advantage of sharing the same memory space. This reduces the complexity of important integration problems, such as how to interpret foreign artefacts in the native language.

However, several integration approaches also target non-embedded logic engines since they usually have access to a considerable amount of well-tested logic libraries and are often more performant than their embedded counterparts.

RQ.2 What are the linguistic integration features required for hybrid logic and object-oriented systems?

In chapter 3 we showed that several techniques for integrating logic and object-oriented systems demand a considerable amount of boilerplate code. This is because the programmer is obliged to explicitly specify every detail of the integration, which is a repetitive and error-prone task. We also observed that some techniques alleviate this problem by attempting an oblivious integration (*i.e.*, the integration is not explicit nor evident in the integrated code). However, these techniques are typically non-portable, since they often require a specific architecture (*e.g.*, the logic engine and the object-oriented program sharing the same memory space).

In chapter 4 we discussed that an oblivious integration approach demands that foreign-language artefacts can be represented as native ones [144]. We also identified the issue of dealing with object state as one of the most critical points when integrating objects with logic [1, 104]. In addition, foreign routines should be transparently invoked as native routines and their outcome, if any, interpreted as native artefacts [66].

These properties determine the degree of obliviousness regarding the integration concern in a hybrid logic and object-oriented system.

RQ.3 From the conceptual point of view, what are the high-level steps in order to accomplish a linguistic integration?

In our conceptual analysis provided in chapter 4, we proposed two general steps for integrating programs at the linguistic level: (1) defining a bidirectional mapping of objects and terms and (2) establishing mappings between routines and their outcomes in both worlds.

We also observed in chapter 5 that the development of these inter-language mappings is greatly facilitated by means of techniques reducing the paradigm mismatch between the two worlds, such as the use of an object-oriented layer in the logic world.

RQ.4 To what extent is it possible to profit from the additional type data provided by a statically-typed language to accomplish an automatic and transparent integration?

In the examples presented in chapter 8 and 9 we made use of static type analysis to guide the conversion of inter-language artefacts. The two scenarios we discussed were: (1) making use of the return type of a method to guide how a term representing a query solution could be converted into an object and (2) making use of the type of a field to determine how a term that should be assigned to that field could be converted into an object.

We also discussed in section 9.4.2 that it was not convenient to attempt to infer conversions of terms to method parameters based on the types

of such parameters. This is because in JAVA a method can be overloaded and multiple definitions of a method with the same name and number of parameters could exist in a class or its super classes. This makes it impossible to determine the correct target type of a term to object conversion.

RQ.5 How can an integration approach be customised to a particular user-defined context?

As discussed in chapters 6, 7 and 8, an inter-language conversion strategy could be scoped to a particular context. Different interactions could refer to distinct contexts thus providing a programmer with more fine-grained control over how the integration should happen in different context-dependent scenarios.

At the moment, our JPC library supports the management of multiple conversion contexts at any given time. However, LOGICOBJECTS supports only one context at a time, since it makes use of a default context (although this context can be changed, only one default context exists at any given time).

10.2 Future Work

In this section we provide insights into possible future work directions.

10.2.1 Inter-Language Reflection

We would like to add constructs and extensions to our library in order to support inter-language reflection [67] (*i.e.*, reflecting the object-oriented language from within the logic language and vice versa). For example, by allowing the querying, refactoring and transformation of JAVA code from PROLOG using linguistic integration techniques. The emergence of approaches targeting this problem [80, 40, 114, 25, 117] is a proof of the interest of the community in such techniques.

10.2.2 Multi-threading support

We are planning to add support for common multi-threading inter-operability operations. LOGTALK's high-level multi-threading features [99, 100] seem a natural choice for solving this problem. This will be the basis for providing high-level support for other concurrency interaction patterns, such as asynchronous interactions, which opens the door to performance improvements. For instance, more efficient mechanisms to obtain all the solutions to a query can be designed, like bringing asynchronously the solutions in chunks, until all solutions are exhausted.

10.2.3 Development of New Drivers

Another future work direction concerns the development of new drivers, both for the currently supported (*e.g.*, by adding new communication protocols such as HTTP) and new PROLOG engines. Particularly, in order to make our approach suitable to real-life problems, we will need to develop a native driver optimised for the JPC and LOGICOBJECTS requirements.

10.2.4 Improvement of the JPC Embedded Prolog Database

In section 6.2.2 we describe that JPC makes use of an internal PROLOG database. This database is currently quite limited and minimalistic. Therefore, it is not intended to be used directly by the programmer.

At the moment of writing, its current version supports assertion and retraction of facts, unification, term indexing and backtracking. It does not support rules, parsing of PROLOG clauses, operators and many other standard and de facto PROLOG features. In the short term, we are counting on adding support for rules and parsing of PROLOG clauses.

Alternatively, we may evaluate replacing this component with an existing PROLOG engine embedded in JAVA with support for object reference terms.

10.2.5 Development of new Case Studies

We will develop new and larger case studies, possibly oriented towards the application of inter-language reflection. We would like to accomplish this in collaboration with other members of the logic-programming community already interested in this topic.

After finishing these case studies and releasing a stable version of our libraries, we are also planning to conduct a user study to evaluate how programmers perceive the tools we have developed.

10.3 Conclusions

This dissertation has proposed both a conceptual model and a concrete implementation of a portable technique for bidirectional linguistic integration between a logic and a statically-typed object-oriented programming language.

Both the conceptual model and its implementation took as a starting point a study of existing techniques for integrating logic and object-oriented programs. Particularly, we profited from cross-fertilisation of ideas from studying existing solutions to this problem in PROLOG, similar logic languages (*e.g.*, SOUL [114]) and even inter-language conversion libraries in other domains (*e.g.*, Google's GSON library [61]).

The tools presented here offer different level of interoperability abstractions. We are aware that the most abstract layers of our approach may not always provide the best solution to all scenarios requiring JAVA-PROLOG interaction. In particular, systems with high performance requirements may prefer to not

use the inter-language conversion facilities we have made available. This is often the case of frameworks attempting to provide higher-level abstractions that simplify the developer's work.

However, our tools do not impose a particular abstraction level. For example, JPC may be used exactly like other integration libraries (like JPL) without further additions, since our conversion constructs are entirely optional. In any case, LOGICOBJECTS and JPC's high-level constructs can often be employed to quickly prototype a system requiring JAVA-PROLOG interoperability. Afterwards, these constructs may be replaced, at performance critical spots, by JPC's lower level API.

Furthermore, the JPC library can serve as a convenient building block for helping architects in building efficiently more sophisticated frameworks (different from LOGICOBJECTS) for integrating JAVA and PROLOG.

In addition, several of JPC's features can be applied to other scenarios. In fact, we have factored out as separate open source projects the modules in charge of (1) creating and managing type and named categorisations¹ and (2) providing a general mechanism for encapsulating, categorising and applying type-guided conversions between objects.²

We hope that both the JPC and LOGICOBJECTS libraries will be useful to the PROLOG and JAVA community. In this spirit, they are, and will remain, open-source software ³.

¹JGUM [21].

²JCONVERTER[20].

³JPC [22]

Bibliography

The references are sorted alphabetically by first author.

- [1] Vladimir Alexiev. Mutable object state for object-oriented logic programming: A survey. Technical Report TR 93-15, Department of Computing Science, University of Alberta, 615 GSB, Edmonton, Alberta T6G 2H1, August 1993.
- [2] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web services. In *Web Services, Data-Centric Systems and Applications*, pages 123–149. Springer Berlin Heidelberg, 2004. ISBN 978-3-642-07888-0. DOI 10.1007/978-3-662-10876-5_5. URL http://dx.doi.org/10.1007/978-3-662-10876-5_5.
- [3] Mutsunori Banbara, Naoyuki Tamura, and Katsumi Inoue. Prolog cafe : A prolog to java translator system. In Masanobu Umeda, Armin Wolf, Oskar Bartenstein, Ulrich Geske, Dietmar Seipel, and Osamu Takata, editors, *INAP*, volume 4369 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2005. ISBN 3-540-69233-9.
- [4] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006. ISBN 1932394885.
- [5] Grady Booch. Object-oriented development. *IEEE Trans. Software Eng.*, 12(2):211–221, 1986.
- [6] A. H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, ACM '86, pages 36–40, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-4743-4. URL <http://dl.acm.org/citation.cfm?id=324493.324538>.
- [7] Dmitri Boulanger and Ulrich Geske. Using Logic Programming in Java Environment (Extended Abstract). Technical Report 10, Knowledge-Based Systems Group, Vienna University of Technology, Austria, 1998.
- [8] Gilad Bracha and William Cook. Mixin-based inheritance. *SIGPLAN Not.*, 25(10):303–311, September 1990. ISSN 0362-1340. DOI 10.1145/97946.97982. URL <http://doi.acm.org/10.1145/97946.97982>.
- [9] Johan Brichau, Kris Gybels, and Roel Wuyts. Towards linguistic symbiosis of an object-oriented and a logic programming language. In *In Jörg*

- Striegnitz, Kei Davis, and Yannis Smaragdakis, editors, Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2002.
- [10] Johan Brichau, Coen De Roover, and Kim Mens. Open unification for program query languages. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, 2007.
- [11] F. Bueno, M. Carro, M. Hermenegildo, R. Haemmerlé, P. López-García, E. Mera, , J.F. Morales, and G. Puebla-(Eds.). The Ciao System. Ref. Manual (v1.14). Technical report, July 2011. Available at <http://ciao-lang.org>.
- [12] William E. Byrd. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. PhD thesis, Indiana University, September 2009.
- [13] M. Calejo. Interprolog: A declarative java-prolog interface. In *EPIA*. Springer, 2001.
- [14] Miguel Calejo. InterProlog: Towards a Declarative Embedding of Logic Programming in Java. In José Júlio Alferes and João Alexandre Leite, editor, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, pages 714–717. Springer, 2004. ISBN 3-540-23242-7. DOI <http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3229&page=714>.
- [15] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. ISSN 0360-0300. DOI 10.1145/6041.6042. URL <http://doi.acm.org/10.1145/6041.6042>.
- [16] Mats Carlsson and Per Mildner. Sicstus prolog – the first 25 years. *CoRR*, abs/1011.5640, 2010.
- [17] Mats Carlsson et al. *SICStus Prolog User’s Manual*. Swedish Institute of Computer Science, release 3 edition, 1995. URL <http://sicstus.sics.se/>. ISBN 91-630-3648-7.
- [18] Sergio Castro. Logicobjects: A portable and extensible approach for linguistic symbiosis between an object-oriented and a logic programming language. In *Proceedings of the International Conference on Logic Programming, Doctoral Consortium (ICLP DC)*, August 2013.
- [19] Sergio Castro. Hydra: An Open-Source Multi-Engine Prolog Query Browser. <https://github.com/java-prolog-connectivity/jpc.common>, 2014.

- [20] Sergio Castro. JConverter: A Java Framework to Encapsulate, Categorise and Apply Type-Guided Conversions between Objects. <http://jconverter.github.com/>, 2014.
- [21] Sergio Castro. JGum: A Lightweight Categorization Framework for Java. <http://jgum.github.com/>, 2014.
- [22] Sergio Castro. JPC: A Java-Prolog integration library inspired by Google's Gson. <http://java-prolog-connectivity.github.com/>, 2014.
- [23] Sergio Castro. LogicObjects: A Linguistic Integration Framework for Java and Prolog. <https://github.com/java-prolog-connectivity/logicobjects/>, 2014.
- [24] Sergio Castro. MapQuery: Querying and Visualizing Geographical Data using Prolog. <https://github.com/java-prolog-connectivity/mapquery/>, 2014.
- [25] Sergio Castro, Coen De Roover, Andy Kellens, Angela Lozano, Kim Mens, and Theo D'Hondt. Diagnosing and correcting design inconsistencies in source code with logical abduction. *Science of Computer Programming: Special Issue on Software Evolution, Adaptability and Variability*, 76(12): 1113–1129, December 2010. ISSN 0167-6423. DOI DOI:10.1016/j.scico.2010.09.001. URL <http://www.sciencedirect.com/science/article/B6V17-511TN7T-1/2/ea5faf2030ad3a65e517e9b8a0c6a2bf>.
- [26] Sergio Castro, Kim Mens, and Paulo Moura. Logicobjects: A linguistic symbiosis approach to bring the declarative power of prolog to java. In *Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, RAM-SE'12, pages 11–16. ACM, June 2012.
- [27] Sergio Castro, Kim Mens, and Paulo Moura. LogicObjects: Enabling Logic Programming in Java through Linguistic Symbiosis. In Kostis Sagonas, editor, *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 7752 of *Lecture Notes in Computer Science*, pages 26–42, Rome, Italy, January 2013. Springer Berlin Heidelberg.
- [28] Sergio Castro, Kim Mens, and Paulo Moura. Customisable handling of java references in prolog programs. In *To appear in the Proceedings of the International Conference on Logic Programming (ICLP)*, July 2014.
- [29] Sergio Castro, Kim Mens, and Paulo Moura. Jpc: A library for categorising and applying inter-language conversions between java and prolog. *Science of Computer Programming: Experimental Software and Toolkits (EST 6) (Submitted)*, 2014.

- [30] Shigeru Chiba. Javassist – a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
- [31] Ugo Chirico. Jiprolog. <http://www.jiprolog.com/documentation.aspx>, September 2012.
- [32] Maurizio Cimadamore and Mirko Viroli. A prolog-oriented extension of java programming based on generics and annotations. In Vasco Amaral, Luis Marcelino, Luís Veiga, and H. Conrad Cunningham, editors, *PPPJ*, volume 272 of *ACM International Conference Proceeding Series*, pages 197–202. ACM, 2007. ISBN 978-1-59593-672-1.
- [33] Maurizio Cimadamore and Mirko Viroli. Integrating java and prolog through generic methods and type inference. In *Proceedings of the 2008 ACM symposium on Applied computing, SAC '08*, pages 198–205, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-753-7. DOI 10.1145/1363686.1363740. URL <http://doi.acm.org/10.1145/1363686.1363740>.
- [34] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA, 2003. ISBN 978-0-201-70372-6.
- [35] Alain Colmerauer. The birth of prolog. In *III, CACM Vol.33, No7*, pages 37–52, 1993.
- [36] International Electrotechnical Commission and International Organization for Standardization. *Information Technology, Programming Languages, Prolog: International Standard. Modules*. Number pt. 2 in International standard ISO: International Organization for Standardization. ISO/IEC, 2000. URL <http://books.google.be/books?id=6vRCmwEACAAJ>.
- [37] Microsoft Corporation. Dcom technical overview. Technical report, Microsoft Corporation, Redmond, WA, November 1996. URL [http://msdn2.microsoft.com/en-us/library/ms809340\(d=printer\).aspx](http://msdn2.microsoft.com/en-us/library/ms809340(d=printer).aspx).
- [38] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog System. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.
- [39] D. Crockford. *JavaScript: The Good Parts: The Good Parts*. O'Reilly Media, 2008. ISBN 9780596554873. URL <http://books.google.be/books?id=PXa2bbY0oQ0C>.
- [40] Coen De Roover and Reinout Stevens. Building development tools interactively using the ekeko meta-programming library. In *Proceedings of the IEEE CSMR-WCRE 2014 Software Evolution Week (CSMR-WCRE14)*, 2014.

- [41] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, January 1998.
- [42] Kris De Volder. JQuery: A generic code browser with a declarative configuration language. In *PADL*, pages 88–102, 2006.
- [43] Bart Demoen and Paul Taurau. jProlog: An Experimental Prolog to Java Compiler. <http://people.cs.kuleuven.be/~bart.demoen/PrologInJava/>, 1996.
- [44] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuprolog: A light-weight prolog for internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-41768-2. DOI 10.1007/3-540-45241-9_13. URL http://dx.doi.org/10.1007/3-540-45241-9_13.
- [45] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm java-prolog integration in tuprolog. *Science of Computer Programming*, 57(2):217 – 250, 2005. ISSN 0167-6423. DOI <http://dx.doi.org/10.1016/j.scico.2005.02.001>. URL <http://www.sciencedirect.com/science/article/pii/S0167642305000158>.
- [46] Maja D’Hondt. *Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality*. PhD thesis, Vrije Universiteit Brussel, May 2004.
- [47] D’Hondt, Maja and Gybels, Kris and Jonckers, Viviane. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC ’04, pages 1328–1335, New York, NY, USA, 2004. ACM.
- [48] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997. ISBN 013215871X.
- [49] Roland Ducournau, Michel Habib, Marianne Huchard, and Marie-Laure Mugnier. Proposal for a Monotonic Multiple Inheritance Linearization. In *OOPSLA*, pages 164–175, 1994.
- [50] M. Easton and J. King. *Cross-Platform .NET Development: Using Mono, Portable.NET, and Microsoft .NET*. Apresspod Series. Springer-Verlag New York Incorporated, 2004. ISBN 9781590593301. URL <http://books.google.be/books?id=059QAAAAAAAJ>.
- [51] Robert Eckstein, Marc Loy, and Dave Wood. *Java Swing*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, 1998. ISBN 1-56592-455-X.

- [52] Nabil A. Elshiewy. Modular and communicating objects in sicstus prolog. In *FGCS*, pages 792–799, 1988.
- [53] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38, October 1997. ISSN 0001-0782. DOI 10.1145/262793.262798. URL <http://doi.acm.org/10.1145/262793.262798>.
- [54] Maydene Fisher, Jon Ellis, and Jonathan Bruce. *JDBC API Tutorial and Reference*. Addison-Wesley, Boston, MA, 2003. ISBN 978-0-321-17384-3.
- [55] Peter Flach. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc., New York, NY, USA, 1994. ISBN 0-471-94152-2.
- [56] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, first edition, 2008. ISBN 9780596516178.
- [57] Ernest Friedman-Hill. *Jess in Action: Java Rule-based Systems*. Manning, Greenwich, CT, 2003. ISBN 1-930-11089-8.
- [58] Carl Friedrich Bolz. Pyrolog: A Prolog interpreter written in Python using the PyPy translator toolchain. <https://bitbucket.org/cfbolz/pyrolog/>, 2009.
- [59] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.
- [60] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.
- [61] Google Inc. Gson 2.2.2: A Java library to convert JSON strings to Java objects and vice-versa. <http://code.google.com/p/google-gson/>, July 2012.
- [62] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.
- [63] David N. Gray, John Hotchkiss, Seth LaForge, Andrew Shalit, and Toby Weinberg. Modern languages and microsoft's component object model. *Commun. ACM*, 41(5):55–65, May 1998. ISSN 0001-0782. DOI 10.1145/274946.274957. URL <http://doi.acm.org/10.1145/274946.274957>.
- [64] Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006. URL <http://www.omg.org/docs/formal/06-04-01.pdf>.

- [65] C. Gülcü. *The Complete Log4j Manual*. QOS.ch, 2003. ISBN 9782970036906. URL <http://books.google.be/books?id=hZBimlxiiyAcC>.
- [66] Kris Gybels. Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. In *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, 2003.
- [67] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Comput. Lang. Syst. Struct.*, 32(2-3):109–124, July 2006. ISSN 1477-8424. DOI 10.1016/j.cl.2005.10.003. URL <http://dx.doi.org/10.1016/j.cl.2005.10.003>.
- [68] Rémy Haemmerlé and François Fages. Modules for prolog revisited. In *Proceedings of the 22Nd International Conference on Logic Programming, ICLP'06*, pages 41–55, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36635-0, 978-3-540-36635-5. DOI 10.1007/11799573_6. URL http://dx.doi.org/10.1007/11799573_6.
- [69] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *Pervasive Computing*, 7(4):12–18, October 2008. ISSN 1536-1268. DOI 10.1109/MPRV.2008.80. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4653466&tag=1.
- [70] Michael Hanus. Multi-paradigm declarative languages. In Véronica Dahl and Ilkka Niemelä, editors, *Logic Programming*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74608-9. DOI 10.1007/978-3-540-74610-2_5. URL http://dx.doi.org/10.1007/978-3-540-74610-2_5.
- [71] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. *SIGPLAN Not.*, 25(10):169–180, September 1990. ISSN 0362-1340. DOI 10.1145/97946.97967. URL <http://doi.acm.org/10.1145/97946.97967>.
- [72] Michi Henning. The rise and fall of corba. *Queue*, 4(5):28–34, June 2006. ISSN 1542-7730. DOI 10.1145/1142031.1142044. URL <http://doi.acm.org/10.1145/1142031.1142044>.
- [73] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *TPLP*, 12(1–2):219–252, 2012. <http://arxiv.org/abs/1102.5497>.
- [74] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-Oriented Programming. *Journal of Object Technology*, March-April 2008, *ETH Zurich*, 7(3):125–151, 2008.

- [75] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. Rbcl: A reflective object-oriented concurrent language without a run-time kernel. In *International Workshop on New Models for Software Architecture (IMSA): Reflection And Meta-Level Architecture*, pages 24–35, 1992.
- [76] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, 2006. ISSN 0164-0925. DOI 10.1145/1152649.1152650.
- [77] Kohsuke Kawaguchi, Joe Fialli, and Sekhar Vajjhala. The java architecture for xml binding (jaxb) 2.2, 2009. URL <http://jcp.org/en/jsr/detail?id=222>.
- [78] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es): 154, 1996. ISSN 0360-0300.
- [79] Gregor Kiczales and Andreas Paepcke. *Open Implementations and Metaobject Protocols*. MIT Press, Cambridge, MA, USA, 1996. URL <http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-TUT95/for-web.pdf>.
- [80] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution*, LATE '07, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-655-4. DOI 10.1145/1275672.1275678. URL <http://doi.acm.org/10.1145/1275672.1275678>.
- [81] Jørgen Lindskov Knudsen. Name Collision in Multiple Classification Hierarchies. In Stein Gjessing and Kristen Nygaard, editors, *ECOOP 88 European Conference on Object-Oriented Programming, Oslo, Norway, August 15-17, 1988, Proceedings*, volume 322 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 1988. ISBN 3-540-50053-7. DOI <http://link.springer.de/link/service/series/0558/bibs/0322/03220093.htm>.
- [82] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. ISBN 0201325772.
- [83] R. Lucas and Inc Quintus Computer Systems. *Application Programming in Quintus Prolog*. Alfred Waller, 1993. ISBN 9781872474045. URL <http://books.google.be/books?id=Bq1QAAAAMAAJ>.
- [84] Tim A. Majchrzak and Herbert Kuchen. Logic java: combining object-oriented and logic programming. In *Proceedings of the 20th international conference on Functional and constraint logic programming*, WFLP'11, pages 122–137, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22530-7. URL <http://dl.acm.org/citation.cfm?id=2032603.2032614>.

- [85] Eve Maler, Jean Paoli, Tim Bray, François Yergeau, and Michael Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. <http://www.w3.org/TR/2008/REC-xml-20081126/>.
- [86] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP*, pages 2–28, 2003.
- [87] Brett McLaughlin. *Java and XML: Solutions to Real-World Problems*. O’Reilly, Sebastopol, CA, 3 edition, 2006. ISBN 978-0-596-10149-7.
- [88] Erik Meijer and Peter Drayton. Static Typing Where Possible, Dynamic Typing When Needed. 2005. URL <http://research.microsoft.com/~{}emeijer/Papers/RDL04Meijer.pdf>.
- [89] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005. ISSN 0360-0300. DOI 10.1145/1118890.1118892. URL <http://doi.acm.org/10.1145/1118890.1118892>.
- [90] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988. ISBN 0136290493.
- [91] Sun Microsystems. Annotations (JDK 5.0 Java Programming Language-related APIs & developer guides), 2008. URL <http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>.
- [92] Miklós Espák. Japlo: Rule-based programming on java. *Journal of Universal Computer Science*, 12(9):1177–1189, sep 2006.
- [93] Paulo Moura. Logtalk web site. <http://logtalk.org/>.
- [94] Paulo Moura. Logtalk 2.6 Documentation. Technical Report DMI 2000/1, University of Beira Interior, Portugal, July 2000.
- [95] Paulo Moura. *Logtalk – Design of an Object-Oriented Logic Programming Language*. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal, September 2003.
- [96] Paulo Moura. Programming patterns for logtalk parametric objects. In Salvador Abreu and Dietmar Seipel, editors, *Applications of Declarative Programming and Knowledge Management*, volume 6547 of *Lecture Notes in Artificial Intelligence*, pages 52–69. Springer-Verlag, Berlin Heidelberg, April 2011.
- [97] Paulo Moura. *Logtalk 3 Reference Manual*, Updated for version 3.00.0 Beta 4 edition, May 2014.
- [98] Paulo Moura. *Logtalk 3 User Manual*, Updated for version 3.00.0 Beta 4 edition, May 2014.

- [99] Paulo Moura, Paul Crocker, and Paulo Nunes. High-Level Multi-threading Programming in Logtalk. In Paul Hudak and David S. Harren, editors, *Proceedings of the Tenth International Symposium on Practical Aspects of Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*, pages 265–281, Berlin Heidelberg, January 2008. Springer-Verlag.
- [100] Paulo Moura, Ricardo Rocha, and Sara C. Madeira. High Level Thread-Based Competitive Or-Parallelism in Logtalk. In Andy Gill and Terrance Swift, editors, *Proceedings of the Eleventh International Symposium on Practical Aspects of Declarative Languages*, volume 5418 of *Lecture Notes in Computer Science*, pages 107–121, Berlin Heidelberg, January 2009. Springer-Verlag.
- [101] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., 2006. ISBN 0596527756.
- [102] Object Technology International, Inc. *Eclipse platform. Technical overview. White Paper*, July 2001.
- [103] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. Artima Incorporation, USA, 2nd edition, 2011. ISBN 0981531644, 9780981531649.
- [104] Andrea Omicini and Antonio Natali. Object-oriented computations in logic programming. In Mario Tokoro and Remo Pareschi, editors, *Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 194–212. Springer Berlin Heidelberg, 1994. ISBN 978-3-540-58202-1. DOI 10.1007/BFb0052184. URL <http://dx.doi.org/10.1007/BFb0052184>.
- [105] Joseph O'Neil and Herbert Schildt. *Javabeans Programming from the Ground Up*. Osborne/ Mc-Graw Hill, 1998. ISBN 007882477X.
- [106] Lukasz Opyrchal, , Lukasz Opyrchal, and Atul Prakash. Efficient object serialization in java. In *In Proceedings of 19th IEEE International Conference on Distributed Computing Systems Workshops (31 May-4, 1998*.
- [107] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, December 1972. ISSN 0001-0782. DOI 10.1145/361598.361623. URL <http://doi.acm.org/10.1145/361598.361623>.
- [108] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, February 2007. ISSN 0018-9162. DOI 10.1109/MC.2007.53. URL <http://dx.doi.org/10.1109/MC.2007.53>.
- [109] Fernando C. N. Pereira and David H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artif. Intell.*, 13(3):231–278, 1980.

- [110] Giulio Piancastelli, Alex Benini, Andrea Omicini, and Alessandro Ricci. The architecture and design of a malleable object-oriented Prolog engine. In Roger L. Wainwright, Hisham M. Haddad, Ronaldo Menezes, and Mirko Viroli, editors, *23rd ACM Symposium on Applied Computing (SAC 2008)*, volume 1, pages 191–197, Fortaleza, Ceará, Brazil, 16–20 March 2008. ACM. ISBN 978-1-59593-753-7. DOI 10.1145/1363686.1363739. URL <http://portal.acm.org/citation.cfm?id=1363686.1363739>. Special Track on Programming Languages.
- [111] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.
- [112] A. Pineda and M. Hermenegildo. O’ciao: An object oriented programming model for (ciao) prolog. Technical Report CLIP 5/99.0, Facultad de Informática, UPM, 1999.
- [113] Tobias Rho, Lukas Degener, Günter Kniesel, Frank Mühlischlegel, Eva Stöwe, Fabian Noth, Andreas Becker, and Ilshat Alyiev. The Prolog Development Tool - A Prolog IDE for Eclipse. <http://sewiki.iai.uni-bonn.de/research/pdt/>, 2004.
- [114] Coen De Roover, Carlos Noguera, Andy Kellens, and Viviane Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *PPPJ*, pages 71–80, 2011.
- [115] Peter Van Roy, editor. *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, volume 3389 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-25079-4.
- [116] S. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 1995.
- [117] Semmle Ltd. SemmleCode. <http://semmle.com/>, 2010.
- [118] SICStus Prolog. Jasper. <http://www.sics.se/sicstus/docs/4.2.1/html/sicstus/Jasper-Overview.html#Jasper-Overview>.
- [119] Paul Singleton, Fred Dushin, and Jan Wielemaker. JPL 3.0: A bidirectional interface between Prolog and Java. http://www.swi-prolog.org/packages/jpl/java_api/, February 2004.
- [120] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O’Reilly & Associates, Inc., Sebastopol, CA, USA, 2002. ISBN 0-596-00095-2.
- [121] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of constraint handling rules. *ACM Trans. Program. Lang. Syst.*, 31(2):8:1–8:42, February 2009. ISSN 0164-0925.

- DOI 10.1145/1462166.1462169. URL <http://doi.acm.org/10.1145/1462166.1462169>.
- [122] Guy L. Steele. *Common LISP: The Language*. Digital Press, Bedford, MA, 2 edition, 1990. ISBN 978-1-55558-041-4.
- [123] Leon Sterling and Ehud Shapiro. *The Art of Prolog (2nd ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-19338-8.
- [124] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000. ISBN 0201700735.
- [125] SWI. SWI-Prolog Users Mailing List. <http://www.swi-prolog.org/Mailinglist.txt>.
- [126] T. Swift and D.S. Warren. XSB: Extending the power of Prolog using tabling. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.
- [127] Antero Taivalsaari. Classes vs. prototypes - some philosophical and historical observations. In *Journal of Object-Oriented Programming*, pages 44–50. SpringerVerlag, 1996.
- [128] Paul Tarau. Jinni: Intelligent mobile agent programming at the intersection of java and prolog. In *In Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, 1999.
- [129] Paul Tarau. Agent Oriented Logic Programming Constructs in Jinni 2004. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 477–478. Springer, 2004. ISBN 3-540-22671-0.
- [130] Paul Tarau. The binprolog experience: Architecture and implementation choices for continuation passing prolog and first-class logic engines. *Theory Pract. Log. Program.*, 12(1-2):97–126, January 2012. ISSN 1471-0684. DOI 10.1017/S1471068411000433. URL <http://dx.doi.org/10.1017/S1471068411000433>.
- [131] Paul Tarau. Styla: a lightweight Scala-based Prolog interpreter based on a pure object oriented term hierarchy. <http://code.google.com/p/styla/>, 2012.
- [132] Paul Tarau and Michel Boyer. Nonstandard answers of elementary logic programs. In *Constructing logic programs*, pages 279–300. John Wiley & Sons, Inc., 1993.

- [133] Bruce Tate and Curt Hibbs. *Ruby on Rails: Up and Running*. O'Reilly Media, Inc., 2006. ISBN 0596101325.
- [134] Thuan L. Thai and Hoang Lam. *.NET Framework Essentials (2Nd Edition)*. O' Reilly & Associates, Inc., 2002. ISBN 0596003021.
- [135] Robbie Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress)*. APress, 2008. ISBN 1590599977, 9781590599976.
- [136] Luke VanderHart and Stuart Sierra. *Practical Clojure*. Apress, Berkely, CA, USA, 1st edition, 2010. ISBN 1430272317, 9781430272311.
- [137] J. Weaver, W. Gao, S. Chin, D. Iverson, and J. Vos. *Pro JavaFX 2: A Definitive Guide to Rich Clients with Java Technology*. Apresspod Series. Apress, 2012. ISBN 9781430268727. URL <http://books.google.be/books?id=Wp9Xrm1ujncC>.
- [138] Jan Wielemaker and Anjo Anjewierden. An architecture for making object-oriented systems available from prolog. In Alexandre Tessier, editor, *WLPE*, pages 97–110, 2002.
- [139] Jan Wielemaker and Anjo Anjewierden. *Programming in XPCE/Prolog*. Universiteit Van Amsterdam, Roetersstraat 15, 1018 WB Amsterdam. The Netherlands, February 2002.
- [140] Jan Wielemaker and Vítor Santos Costa. On the portability of prolog applications. In *PADL*, pages 69–83, 2011.
- [141] Jan Wielemaker and Michael Hendricks. Why it's nice to be quoted: Quasiquoting for prolog. *CoRR*, abs/1308.3941, 2013.
- [142] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. ISSN 1471-0684.
- [143] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.
- [144] Roel Wuyts and Stéphane Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. *International Workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.
- [145] Guizhen Yang, Michael Kifer, and Chang Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *In Second International Conference on Ontologies, Databases and Applications of Semantics (ODBASE)*, 2003.
- [146] YAP. YAP-Prolog Users Mailing List. <https://lists.sourceforge.net/lists/listinfo/yap-users>.

List of Terms

- Logic Programming (LP), 5
- Object-Oriented Programming (OOP), 5
- abstraction approach, 2
- annotations, 26
- bidirectional integration, 43
- bindings, 14
- Black Box Logic Reification, 75
- case classes, 80
- categories, 37
- causal connection, 73
- class, 37
- closed world assumption, 38
- computation, 5
- Constant Unification of Object References, 75
- definite clause grammar, 17
- drivers, 187
- engine space, 126
- Equality Preserving Reification, 76
- Explicit Reference Life Span, 77
- Garbage Collected Reference Life Span, 77
- Identity Preserving Reification, 76
- instance, 37
- Integration Policy, 90
- inter-language reflection, 30
- linguistic symbiosis, 1
- logic, 5
- logic layers, 30
- metaclass, 37
- objects, 37
- open implementations, 30
- open unification, 30, 75
- Open Unification of Object References, 76
- paradigm leaking, 5
- paradigm mismatch, 26, 72
- paradigmatic buffer, 72
- parametric object, 38
- parent, 37
- program, 5
- protocols, 37
- prototype, 37
- query solution, 14
- Reference Logic Reification, 74
- separation of concerns, 42
- soft references, 78
- Strong Reification of a Reference, 78
- subclass, 37
- superclass, 37
- Symbiotic Class, 85
- Symbiotic Method, 85
- symbiotic methods, 27
- Symbiotic Module, 85
- Symbiotic Predicate, 85
- Symbolic Logic Reification, 74
- type parameters, 26
- type system, 6
- unidirectional integration, 43
- weak references, 78
- Weak Reification of a Reference, 78
- White Box Logic Reification, 75
- wildcard types, 26

Acronyms

JPC	Java–Prolog Connectivity
JDBC	Java–Database Connectivity
AST	Abstract Syntax Tree
OOP	Object-Oriented Programming
LP	Logic Programming
AOP	Aspect-Oriented Programming
JVM	Java Virtual Machine
PVM	Prolog Virtual Machine
ISO	International Organization for Standardization
OSM	Open Street Map
SAX	Simple API for XML