Dissertation

Integration of Prolog and Java with the Connector Architecture CAPJa

Ludwig Ostermayer

February 2017

Supervisor:  Prof. Dr. Dietmar Seipel

Referees:    Prof. Dr. Dietmar Seipel
              Prof. Dr. Grzegorz J. Nalepa
To Bernadette
Abstract

Modern software is often realized as a modular combination of subsystems for, e.g., knowledge management, visualization, verification, or the interaction with users. As a result, software libraries from possibly different programming languages have to work together. Even more complex the case is if different programming paradigms have to be combined. This type of diversification of programming languages and paradigms in just one software application can only be mastered by mechanisms for a seamless integration of the involved programming languages. However, the integration of the common logic programming language PROLOG and the popular object-oriented programming language JAVA is complicated by various interoperability problems which stem on the one hand from the paradigmatic gap between the programming languages, and on the other hand, from the diversity of the available PROLOG systems.

The subject of the thesis is the investigation of novel mechanisms for the integration of logic programming in PROLOG and object-oriented programming in JAVA. We are particularly interested in an object-oriented, uniform approach which is not specific to just one PROLOG system. Therefore, we have first identified several important criteria for the seamless integration of PROLOG and JAVA from the object-oriented perspective. The main contribution of the thesis is a novel integration framework called the Connector Architecture for Prolog and Java (CAPJa). The framework is completely implemented in JAVA and imposes no modifications to the JAVA Virtual Machine or PROLOG. CAPJa provides a semi-automated mechanism for the integration of PROLOG predicates into JAVA. For compact, readable, and object-oriented queries to PROLOG, CAPJa exploits lambda expressions with conditional and relational operators in JAVA. The communication between JAVA and PROLOG is based on a fully automated mapping of JAVA objects to PROLOG terms, and vice versa. In JAVA, an extensible system of gateways provides connectivity with various PROLOG system and, moreover, makes any connected PROLOG system easily interchangeable, without major adaption in JAVA.
Kurzbeschreibung


Acknowledgements

First of all, I would like to thank my mentor Professor Dr. Dietmar Seipel for introducing me to the wonderful world of logic programming, for his wise guidance, continuous support, and incredible patience without which this thesis would not have been possible. I also thank Professor Dr. Grzegorz J. Nalepa for countless valuable suggestions, important remarks, and fruitful discussions which had major impact on the work at hand. Moreover, I thank my colleague Frank Flederer for the close collaborations in the early stages of CAPJa and Dr. Joachim Spoerhase for his advice during the completion of this thesis. I also would like to express my thanks to Dr. Ulrike Rapp–Galmiche for proofreading this thesis. Finally, I would like to thank all students that have been involved in CAPJa in any manner, most notably Mirco Lukas and Andrew Easton. Last but not least, I would like to express my gratitude to all my friends and my family for their consistent support and encouragement, most especially my dearest Bernadette.
## Contents

1 Introduction
   1.1 Motivation, Scope, and Research Problem .......................... 1
   1.2 Goal of the Work and Original Contribution ........................ 3
   1.3 Introductory Examples .............................................. 6
   1.4 Plan of the Work ................................................... 8
   1.5 Exclusions .......................................................... 9
   1.6 Supporting Publications ............................................ 9

2 Selected Programming Paradigms and Languages .................. 11
   2.1 The Logic Programming Paradigm .................................. 12
      2.1.1 The Language of First-Order Logic .......................... 12
      2.1.2 Theory of Unification ....................................... 15
      2.1.3 The Computational Model .................................... 18
   2.2 The Logic Programming Language PROLOG .......................... 21
      2.2.1 Characteristics ................................................. 21
      2.2.2 Important Language Concepts and Artifacts ................. 23
   2.3 The Object-Oriented Programming Paradigm ....................... 27
      2.3.1 Objects and Classes ......................................... 27
      2.3.2 Information Hiding and Encapsulation ........................ 29
      2.3.3 Types and Subtypes .......................................... 29
      2.3.4 Inheritance and Polymorphy .................................. 30
      2.3.5 Dynamic Method Dispatch .................................... 31
   2.4 The Object-Oriented Programming Language JAVA ................... 31
      2.4.1 Characteristics ................................................. 31
      2.4.2 Important Language Concepts and Artifacts ................. 32
   2.5 Synergy of Prolog and Java ........................................ 35
      2.5.1 Advantages and Disadvantages of PROLOG ..................... 36
      2.5.2 Advantages and Disadvantages of JAVA ...................... 37
      2.5.3 Synergy Effects ............................................... 39

3 Integration Approaches for Prolog and Java ..................... 41
   3.1 Integration via Translation ........................................ 41
      3.1.1 jProlog ......................................................... 41
Contents

3.1.2 Prolog Cafe .................................. 42
3.2 Integration via Embedding .......................... 45
  3.2.1 tuProlog and P@J .............................. 46
  3.2.2 Smalltalk and SOUL ......................... 49
  3.2.3 Logic Java .................................. 51
3.3 Integration via Communication Interfaces ............. 54
  3.3.1 Interprolog .................................. 54
  3.3.2 JPL ........................................ 57
  3.3.3 PBR4J ....................................... 60
  3.3.4 LogicObjects ................................ 63
3.4 Alternative Approaches ............................ 67
3.5 Synthesis ........................................ 68
  3.5.1 Discussion .................................. 69
  3.5.2 Important Criteria for a Seamless Integration .... 71

4 The Connector Architecture CAPJa .................... 73
  4.1 Components .................................... 74
  4.2 Development with CAPJa ......................... 78

5 The JPMapping Component ............................ 81
  5.1 Object-To-Term Mappings ......................... 82
    5.1.1 Introductory Example ....................... 82
    5.1.2 Formal Definition .......................... 83
    5.1.3 Default Mapping Mechanism .................. 86
    5.1.4 Custom Mapping Definitions ................. 87
  5.2 Predicate-Signature Notation ...................... 91
    5.2.1 Introductory Example ....................... 92
    5.2.2 Syntax and Semantics ....................... 93
  5.3 Implementation of Mappings ...................... 95
    5.3.1 The Abstract JPMapper<T> Class ............. 95
    5.3.2 Subclassing JPMapper<T> .................... 96
    5.3.3 Source Code Generation ..................... 100

6 The JPLambda Component .............................. 105
  6.1 Introductory Examples ........................... 106
  6.2 The Query Language JPQL ......................... 109
    6.2.1 Lambda Expressions ........................ 109
    6.2.2 Query Constraints .......................... 110
    6.2.3 Query Execution ............................ 112
  6.3 The Mapping Language JPML ....................... 114
6.4  JPCompiler ................................................. 116
  6.4.1  Workflow ............................................. 117
  6.4.2  Source Code Parsing ................................. 119
  6.4.3  The JPLambdaAnalyzer Class ....................... 120
  6.4.4  Generated Source Code .............................. 121
  6.4.5  Modified Source Code .............................. 123

7  The JPGateway Component .............................. 125
  7.1  The JPGateway Interface ............................. 125
  7.2  Dependencies .......................................... 127
  7.3  The Portable Prolog Gateway ......................... 129
  7.4  Custom Gateways ...................................... 134
    7.4.1  Remote Procedure Calls ............................ 134
    7.4.2  CAPJa as a High-Level API ........................ 139

8  Evaluation .............................................. 143
  8.1  Case Study on the London Underground ................ 143
    8.1.1  The Prolog Knowledge Base ....................... 144
    8.1.2  Queries from JAVA to PROLOG ...................... 145
    8.1.3  Implementation with CAPJa ....................... 147
    8.1.4  Implementation with JPL ......................... 148
    8.1.5  Results ............................................ 150
  8.2  Case Study on Taxes in International E-Commerce .... 153
    8.2.1  Logic Programming for Business Rules .......... 153
    8.2.2  Delivery Threshold ............................... 155
    8.2.3  Business Objects .................................. 155
    8.2.4  Business Rules ................................... 158
    8.2.5  Calling the Business Rules ...................... 160
    8.2.6  Results ............................................ 162
  8.3  Validation ............................................ 163
    8.3.1  Applicability Revisited ......................... 163
    8.3.2  Important Criteria for a Seamless Integration Revisited .... 165

9  Conclusion .............................................. 173
  9.1  Summary ................................................ 173
  9.2  Future Work ........................................... 176
List of Figures

3.1 JFrame with 4 Buttons ........................................ 60
4.1 The Connector Architecture for PROLOG and JAVA. .......... 74
4.2 The JPMAPPING component of CAPJa. .......................... 76
5.1 The annotation layer of JPMAPPING. ............................... 88
5.2 The abstract JPMapper<T> class and its abstract methods. ... 95
5.3 Generating of subclasses of JPMapper<T> with JPMappingProcessor. 101
5.4 Generation of the object-oriented interface from Psn............ 102
6.1 The Employee and the Address class in JAVA. ................... 107
6.2 Extended build process with JP_COMPILER. ........................ 116
6.3 Workflow of JP_COMPILER. ...................................... 118
7.1 The JPGATEWAY component and its dependencies within CAPJa. 128
7.2 Communication with PROLOG via standard streams. .......... 129
7.3 Information flow in the PPG. .................................... 130
8.1 An invoice in JAVA. .............................................. 156
8.2 A booking in JAVA. ............................................... 156
8.3 Visualization of the program execution with proof trees. ...... 162
List of Tables

5.1 Java-Prolog type conversion. ........................................ 86

7.1 Implementation of JPPengine based on Prolog API of Pengines. 138
7.2 Implementation of JPLGateway through Query of JPL. ............. 140
7.3 Implementation of IPGateway through PrologEngine of Interprolog. .............................................................. 142

8.1 Necessary LoC with CAPJa and JPL. ................................. 150
8.2 Performance of the PPG, JPLGateway, and JPL with Swi-Prolog. 152
8.3 Performance of the PPG with various Prolog systems. ............. 152
## Source Code Listings

1.1 Some person/3 facts in PROLOG. ......................................... 6  
1.2 Signature of the person/3 facts in Psn. .............................. 6  
1.3 Querying PROLOG from JAVA with CAPJa. .......................... 7  
1.4 Querying PROLOG with additional query constraints. .......... 7  
1.5 Accessing important PROLOG meta-predicates from JAVA with CAPJa. 7  

3.1 A simple PROLOG program for the translation with PROLOG CAFE. 42  
3.2 The transformed binary clauses. ...................................... 42  
3.3 The translated PROLOG program in JAVA. .......................... 43  
3.4 Calling JAVA from PROLOG with PROLOG CAFE. ................. 44  
3.5 Instantiation of a simple PROLOG goal wrapper with P@J. ...... 47  
3.6 A PROLOG class in P@J. ............................................. 48  
3.7 A simple rule in SOUL. .............................................. 50  
3.8 Solving Fermat’s Last Theorem with LOGIC JAVA. ............... 53  
3.9 A raw term in INTERPROLOG for an Integer instance. .......... 55  
3.10 Usage of the java/3 predicate in INTERPROLOG. ............... 56  
3.11 Calling PROLOG from JAVA with INTERPROLOG. ................. 56  
3.12 Calling PROLOG from JAVA with JPL. ............................ 58  
3.13 Calling JAVA from PROLOG with JPL. ............................ 59  
3.14 Foreign Key Constraints. .......................................... 62  
3.15 Fragment of the XML Schema describing tax/2. ................ 62  
3.16 A JAVA call to the business rules in PROLOG. .................. 63  
3.17 The symbiotic class Line in LOGICOBJECTS. .................... 65  
3.18 The parametric line object in LOGTALK. .......................... 65  

5.1 Sample data structures in JAVA. ................................... 82  
5.2 Instance bart of Person in JAVA. .................................. 83  
5.3 Representation of the bart object as term in PROLOG. .......... 83  
5.4 Custom mapping definitions for the Person class in JAVA. .... 90  
5.5 Person instances for the mapping to PROLOG. .................... 90  
5.6 Different representations of the homer object as term in PROLOG. 90  
5.7 A book/6 fact in PROLOG. ......................................... 92  
5.8 Signature of the author/6 predicate in Psn. ........................ 92  
5.9 Signature of member/2 in Psn. .................................... 94
5.10 An implementation of the getInstanceFromBindings method.
5.11 PROLOG grammar for ANTLR.
5.12 An implementation of the getInstanceFromInstance method.
5.13 An implementation of the isMappingType method.
5.14 The classes Book and Author in JAVA.

6.1 Querying PROLOG with JPLAMBDA.
6.2 Corresponding query in PROLOG.
6.3 Instantiating a custom mapper for the Employee class.
6.4 Exemplary target term in PROLOG for employeeMapper.
6.5 Simple lambda expression in JAVA.
6.6 Simple lambda in JPQl with a single query constraint.
6.7 Executing a query to PROLOG in JPQl.
6.8 A query with one query constraint not supported by JPQl.
6.9 Implementations of the abstract methods of JPQueryTranslator<T>.
6.10 Injecting a reference to the generated JPQueryTranslator1 subclass.

7.1 A single entry of the configuration file of the PPG.
7.2 The connect method of PPGateway.
7.3 Implementation of asserta of PPGateway.
7.4 Implementation of backtrack of PPGateway.
7.5 Creation of a PROLOG server based on TCP sockets.
7.6 Opening and requesting a socket from JAVA.
7.7 A simple handler implementation in SWI-PROLOG for POST requests.
7.8 POST-Request from JAVA.
7.9 JAVA methods of JPl for calling PROLOG.
7.10 JAVA methods of INTERPROLOG for calling PROLOG.

8.1 Some PROLOG facts about the London underground.
8.2 Signature of the fact base.
8.3 The predicate reachable/4 in PROLOG.
8.4 Signature of the reachable/3 predicate in PSN.
8.5 The Tour class in JAVA.
8.6 Queries to the London Underground with CAPJa in JAVA.
8.7 Translation methods for the Station class with JPl.
8.8 Query q1 with JPl in JAVA.
8.9 Query q2 with JPl in JAVA.
8.10 Query q3 with JPl in JAVA.
8.11 Custom mapping definitions with JPML.
8.12 Facts provided by JAVA.
8.13 Facts for the configuration in DATALOG∗.
8.14 Aggregation of the current business volume. 158
8.15 Business rules for the delivery threshold. 159
8.16 Bookings corresponding to a position of an invoice. 159
8.17 Accessing the business rules in DATALOG* from JAVA. 160
1 Introduction

This chapter gives an introduction to the work at hand. Section 1.1 describes the background of our research and explains the identified research problems and their importance in the field of computer science. Section 1.2 specifies the main goal of this thesis and outlines the original contributions for which Section 1.3 gives introductory examples. Section 1.4 describes the plan of the work at hand and outlines each subsequent chapter. Section 1.5 shortly discusses issues deliberately not addressed in this work and Section 1.6 concludes the chapter with a list of supporting publications.

1.1 Motivation, Scope, and Research Problem

Modern software is often realized as a modular combination of subsystems for, e.g., knowledge management, visualization, verification, or the interaction with users. As a result, software libraries from possibly different programming languages have to work together. Even more complex the case is if different programming paradigms have to be combined. This type of diversification of programming languages and paradigms in just one software application can only be mastered by mechanisms for a seamless integration of the involved programming languages. However, the integration of the common logic programming language PROLOG and the popular object-oriented programming language JAVA is complicated by various interoperability problems which stem on the one hand from the paradigmatic gap between the programming languages, and on the other hand, from the diversity of the available PROLOG systems.

The object-oriented programming paradigm is widely used in the field of industrial software engineering as well as in the academic sector. Currently, one of the most popular object-oriented programming languages is JAVA. It has a rich set of libraries, especially for refined graphical user interfaces (GUI), web development and embedded devices. In addition, there are many mature and public tools and integrated development environments such as Eclipse [27], that substantially support the development with JAVA. Last but not least, JAVA has a very active community. But there are also issues for JAVA known such as being very verbose, full of boilerplate code, and thus ill-suited for rapid prototyping. Moreover, JAVA has no support for rules,
1 Introduction

Logic programming, instead, is following an alternative and declarative programming paradigm. Logic programming languages such as PROLOG are particularly well suited for the development of rule-based systems, the rule formalism is an essential part of PROLOG. Programs in PROLOG consist of a collection of rules and facts that describe Horn clauses. They are usually evaluated in a top-down manner. PROLOG is widely used in knowledge engineering, expert systems [10], business rule applications [68], and natural language processing [88]. Backtracking, partial bindings, and incomplete structures allow for elegant, concise and very readable programs which are easy to maintain and refactor. Therefore, PROLOG is also well suited for rapid prototyping in agile software development. Custom operators, infix notation, and definite clause grammars allow for readable domain-specific languages (Dsl) [31] and declarative business rules [66]. Because PROLOG is homoiconic, PROLOG programs can easily be inspected which allows for powerful meta-interpreters. Furthermore, PROLOG can be used to build, update, and test complex structured data such as ontologies [6], XML documents [83], or graphs [69].

To this end, it is clear that a synergy of PROLOG and JAVA is desirable. The combination of the strengths of both programming languages and their associated programming paradigms offers novel approaches in software engineering. However, essential for a beneficial synergy are elaborate integration approaches. In particular, the integration of PROLOG and JAVA is complicated by various interoperability problems. Several approaches [37, 26, 15, 89, 2, 5, 21, 50, 17] for the integration of PROLOG and JAVA have been proposed during the last decade and thus prove an ongoing interest and also an actual need for further improvements. A main issue clearly is the compatibility of former issues. Almost all former solutions are designed for a specific PROLOG system. As a result, an integrated PROLOG system cannot be exchanged easily, without major adaption in JAVA. Therefore, we need an uniform approach for the integration of PROLOG and JAVA that is not specific to a particular PROLOG system. Because JAVA has a predominant role in industry and academics, the object-oriented perspective of the integration is of utter importance. However, especially the object-oriented perspective of former approaches lacks accessibility. For instance, queries to PROLOG either are not object-oriented [15] or obfuscated by boilerplate code [89, 21]. Another problem is that user-defined JAVA types cannot
be used directly for queries to PROLOG. Objects in JAVA often have to be manually translated to terms in PROLOG which reduces the readability of programs and considerably increases the necessary programming effort. Moreover, the integration of PROLOG predicates into JAVA is a repetitive and laborious manual task with former approaches which frequently leads to annoying errors. Next to compatibility, accessibility, and programming effort, there are other important criteria for the seamless integration of PROLOG and JAVA from the object-oriented perspective, such as flexibility and productivity.

1.2 Goal of the Work and Original Contribution

The subject of this thesis is the investigation of novel mechanisms for the integration of logic programming in PROLOG and object-oriented programming in JAVA. The main goal of this thesis is an object-oriented, uniform integration approach for PROLOG and JAVA which is not specific to just one PROLOG system. Therefore, we have first identified several important criteria for a seamless integration of PROLOG and JAVA from the object-oriented perspective.

The main contribution of the thesis is a novel integration framework called the Connector Architecture for Prolog and Java (CAPJa). The framework is completely implemented in JAVA and imposes no modifications to the JAVA Virtual Machine or PROLOG. CAPJa has been designed to simultaneously solve the known interoperability problems between PROLOG and JAVA while satisfying at the same time the identified criteria for a seamless integration of PROLOG and JAVA. The main features of CAPJa can be summarized as follows:

- Objects in JAVA can be automatically transformed into terms in PROLOG, and vice versa.
- Predicates in PROLOG can be semi-automatically integrated into JAVA.
- The object-to-term mapping mechanism is highly customizable.
- The interface to PROLOG is purely object-oriented in JAVA.
- The query format for PROLOG is clear and concise due to usage of lambda expressions with conditional and relational operators in JAVA.
- Queries to PROLOG are independent from the connected PROLOG implementation and even almost free of any implications to PROLOG.
- A system of gateways allows to establish different communication channels between PROLOG and JAVA.
A default gateway connects Java with various different Prolog implementations.

In the following, we describe our approach with CAPJa in greater detail. CAPJa is completely implemented in Java and imposes no modifications to the Java Virtual Machine or Prolog. Integrated Prolog predicates can easily be queried from Java with the help of an object-oriented interface that associates generated Java classes with corresponding Prolog predicates. The query format for Prolog is clear and concise due to Boolean lambda expressions with conditional and relational operators in Java. Queries with CAPJa are independent from a specific Prolog system. For this purpose, CAPJa provides an extensible system of gateways to Prolog which separates Java from Prolog and encapsulates the specifics of a given particular Prolog system, similar to the Open Database Connectivity (ODBC) as a standardized programming interface to database management systems. Once the object-oriented interface to Prolog is fixed, the Java part as well as the Prolog part of a multi-paradigm program with CAPJa can be developed independently from each other. In this way, predicates in Prolog can be integrated into Java without paradigmatic interference. CAPJa’s system of gateways is bundled with a default gateway to Prolog, the Portable Prolog Gateway (PPG). The PPG has been designed to work independently from a specific Prolog system and thus enables programs developed with CAPJa to run on different Prolog systems. The PPG has already been tested successfully with BProlog, the Ciao system, GNU-Prolog, SWI-Prolog, Tu-Prolog, the XSB system, and YAPROLOG. In the following, we take a closer look at the features of CAPJa.

To integrate Prolog predicates into Java, CAPJa provides the Predicate-Signature Notation (PSN) in Prolog. The PSN allows us to express in a compact and natural way the signatures of predicates in Prolog. CAPJa uses a source-code generator to create two Java classes from a given predicate signature. The first class associates the predicate in Prolog for which the signature is given in PSN; each argument of the predicate thereby is represented as class member. Therefore, the first class forms an object-oriented interface to Prolog. The second class generated by CAPJa handles the translation of instances of the first class to Prolog, and vice versa. The second class is only used internally by CAPJa.

From the Java perspective, instances of preexisting and user-defined Java types can have a representation as terms in Prolog, too. Therefore, CAPJa provides a compact class annotation layer (@JPMapping) that defines the specification for a mapping of class instances to Prolog. However, if the source code of a class is not accessible for annotations, the Java-Prolog Mapping Language (Jpml) of CAPJa can be used alternatively to define a custom object-to-term mapping. The Jpml is
1.2 Goal of the Work and Original Contribution

a small, in JAVA embedded, domain-specific language (DSL) [31] that exploits lambda expressions for concise definitions in JAVA.

To query PROLOG from JAVA, CAPJa provides the Java-Prolog Query Language (JpQL) which is also an embedded DSL in JAVA. JpQL allows us to specify a query type in JAVA for which a lambda expression can used to define additional query constraints associated to the members of the query type. Almost any common JAVA type is a suitable query type, whereby the types of the object-oriented interface to PROLOG are preferable. Due to the used lambda expressions queries in JpQL are compact, declarative, and purely object-oriented. The return of a query to PROLOG is an instance of the query type. In this way, we avoid the problem of PROLOG’s multiplicity of predicates with no fixed argument for the output. JpQL offers different methods for the evaluation of queries in PROLOG; e.g., we can ask for the existence of a solution, the first solution, or all possible solutions. Furthermore, it is possible to control PROLOG’s backtracking mechanism directly from JAVA. In this way, individual solutions can be obtained one by one via backtracking in PROLOG.

The two embedded DSLs, JpQL and JpML, have no runtime behavior, they just serve as specification languages for queries and custom object-to-term mappings. CAPJa analyzes expressions in JpML or JpQL and generates new JAVA source files that efficiently implement the stated queries and mappings. The original source files with the DSLs are modified by references to the generated classes in the form of additional constructor or method parameters. In this way, we achieve an optimal performance at runtime and CAPJa avoids costly reflection. The complete process is provided as an extended build process for JAVA which does not affect the used JAVA compiler. The application of the build process is only required once, just before a program with CAPJa is executed for the first time.

The overall applicability of our approach with CAPJa is verified by two detailed case studies. In the first case study, we create a query system for the network of the London underground. Therefore, we have built for Swi-PROLOG [99] a knowledge base from data in GraphML, an XML-based file format for graphs. We use CAPJa to integrate the PROLOG knowledge base into JAVA. In this case study, we are interested how CAPJa’s integration and query mechanism perform. For a better comparison, a reference implementation with JPL [89], the custom JAVA-PROLOG interface for Swi-PROLOG, is considered, too. We analyze for both approaches the implementation efforts and how they scale with the complexity of queries. Furthermore, we are interested in how CAPJa’s abstraction layer affects the performance at runtime. In the second case study, we investigate how business rules can be integrated into JAVA. We want to implement the business rules in DATALOG * [87] which is an extension to standard DATALOG and allows for a larger set of connectives (including conjunction
and disjunction), for function symbols, and for stratified Prolog meta-predicates (including aggregation and default negation) in rule bodies. Datalog* programs are evaluated bottom-up, just like standard Datalog programs. Just like the integration of Prolog programs into Java, CAPJa facilitates the integration of Datalog programs into Java, too. For this purpose, we will derive with the help of CAPJa a suitable fact base in Datalog* from preexisting Java types. The derived fact base then forms the starting point for the development of the business rules in Datalog*. Once the object-oriented interface to Datalog* is determined, the business rules can be developed independently from Java. The Java objects then can be asserted directly in Datalog* and thus provide the fact base necessary for the evaluation of the business rules in Datalog*.

To bridge the gap between business analysts and programmers, the business rules have a concise and declarative format and their execution in Datalog* is visualized by proof trees. We conclude our evaluation of CAPJa with a comparison to the discussed related work and the previously derived criteria for a seamless integration.

1.3 Introductory Examples

In the following, we illustrate a typical use case for the development with CAPJa. Given is a simple fact base in Prolog. Each fact represents a person by a first name, last name, and year of birth. Listing 1.1 shows three sample facts.

```
1 % person(+First_Name, +Last_Name, +Year_Of_Birth)
4 person('Pete', 'Sampras', 1971).
```

Listing 1.1: Some person/3 facts in Prolog.

To access the person/3 facts from Java, we first use in Prolog the Predicate-Signature Notation (PSN) to describe their signature, similar to an XML Schema that describes the structure of an XML document. Listing 1.2 shows the signature of the person/3 predicate in PSN.

```
1 % predicate(+Name, +Type, +Arguments)
2 predicate(person/3, compound, [argument(firstName, atom),
3   argument(lastName, atom), argument(yearOfBirth, integer)]).
```

Listing 1.2: Signature of the person/3 facts in PSN.

From the signatures in PSN, CAPJa generates the Person type in Java which we use to access the Prolog fact base. For this purpose, CAPJa provides various methods, as shown in Listing 1.3, to execute a query from Java to Prolog. A call of
person(X,Y,Z) in PROLOG corresponds to `simpleQuery` in Listing 1.3. The special iterator [34] instance of Line 5 in Listing 1.3 offers control over backtracking in PROLOG. Each invocation of its `hasNext` method requests backtracking in PROLOG.

```java
JPQuery<Person> simpleQuery = new JPQuery<Person>();
boolean hasSolution = simpleQuery.hasSolution(gateway);
Person firstPersonFound = simpleQuery.getSolution(gateway);
List<Person> personList = simpleQuery.getAllSolutions(gateway);
Iterator<Person> it = simpleQuery.getLazySolutionsIterator(gateway);
while(it.hasNext()){ // successful backtracking in Prolog...
    Person p = it.next(); // ...retrieves the next solutions
}
```

Listing 1.3: Querying PROLOG from Java with CAPJa.

CAPJa allows for far more complex queries to PROLOG. We use Boolean lambda expressions together with conditional and relational operators in JAVA to express additional query constraints which any solution has to satisfy. Listing 1.4 illustrates the usage of lambdas with CAPJa.

```java
JPQuery<Person> complexQuery = new JPQuery<Person>(
    person -> person.getYearOfBirth > 1981 && person.getFirstName() == "Baker");
```

Listing 1.4: Querying PROLOG with additional query constraints.

The query constraints of the `complexQuery` in Listing 1.4 are not evaluated in JAVA. They are translated to variable bindings and subgoals of the actual query in PROLOG. A query in PROLOG corresponding to `complexQuery` can be written as follows.

```prolog
```

Moreover, any PROLOG predicates can be integrated into JAVA and be accessed in the same way as described above. Furthermore, CAPJa offers additional methods to modify directly PROLOG’s internal database and to load PROLOG files.

```java
Person p = new Person("Bob", "Ross", 1942);
gateway.asserta(p);
gateway.assertz(p);
gateway.retract(p);
gateway.retractAll(p);
gateway.<Person>retractAll(person -> person.getFirstName == "Bob");
gateway.consult("C:/Program Files (x86)/XSB/lib/lists.P");
```

Listing 1.5: Accessing important PROLOG meta-predicates from JAVA with CAPJa.

It is worth mentioning that all examples from above contain almost no code that is specific to PROLOG or specific to a particular PROLOG system. The only implicit
1 Introduction

reference is the **gateway** object which is an abstraction of a connected **Prolog**
database.

1.4 Plan of the Work

Chapter 2 and 3 define the background of the research problem and constitute the
motivation for this thesis.

- **Chapter 2** considers the relevant two programming paradigms and languages.
  We begin with a short introduction to the logic programming paradigm and
  a characterization of the logic programming language **Prolog**. Thereafter,
  we specify the object-oriented programming paradigm and the object-oriented
  programming language **Java**. The chapter concludes with a discussion on the
  synergy of **Prolog** and **Java**.

- **Chapter 3** describes important related work on the integration of logic pro-
  gramming and object-oriented programming, or rather the integration of
  **Prolog** and **Prolog**. Finally, we derive important criteria that facilitate the in-
  tegration of **Prolog** and **Java** from the object-oriented perspective.

Chapter 4, 5, 6, and 7 present the original contributions of this thesis.

- **Chapter 4** gives an overview on **CAPJa** and introduces each of its components
  shortly. This is followed by typical development scenarios and phases with
  **CAPJa**.

- **Chapter 5** introduces the **JPMapping** component of **CAPJa** that provides
  a generic and customisable mapping mechanism for objects in **Java** and terms
  in **Prolog**. An object-to-term mapping can be expressed in **Java** with the
  help of a special annotation layer. For the integration of **Prolog** predicates
  in **Java**, **JPMapping** offers the **PsN**, a small domain-specific language based
  on lists in **Prolog**, which allows us to describe the signature of predicates in
  **Prolog**. It is used to generate classes in **Java** which map to the predicates
  described by predicate signatures in **PsN**.

- **Chapter 6** introduces the **JPLambda** component of **CAPJa** which provides
  the **Jpql** (**Java-Prolog** Query Language) in **Java** that is based on lambda
  expressions. The source code of queries is analyzed by a built-in source-to-
  source compiler that splits a lambda into code that is executed in **Java** and
  into code that is translated to plain **Prolog** code which is transmitted as goal
  to **Prolog**.
Chapter 7 describes the JPGATEWAY component of CAPJa. It starts with a description of CAPJa's interface to any PROLOG gateway. This is followed by an overview of the dependencies in CAPJa for queries to PROLOG. Then, the default gateway of CAPJa is introduced which is not specific to a given PROLOG system. Thereafter, other gateways for the use with CAPJa are presented.

Chapter 8 evaluates our approach with CAPJa. We begin with two detailed case studies. The first case study is about railway networks and the second case study is about the integration of business rules for international e-commerce into JAVA. In both case studies, we demonstrate applications of CAPJa and how CAPJa facilitates the integration of PROLOG and JAVA. Finally, we discuss the overall applicability of CAPJa and compare it to the previously stated criteria for a seamless integration of PROLOG and JAVA.

Chapter 9, finally, concludes this dissertation with a summary of the work at hand and an outline of conceivable future work.

1.5 Exclusions

In consulting related work, we have found that accessing PROLOG from JAVA has always been described as much more complex than the opposite direction. The large number of approaches verify that the associated issues have still not been solved with a satisfactory quality. In this thesis, we focus on the accessing of PROLOG from JAVA and novel mechanisms for PROLOG queries from JAVA, the opposite direction of accessing JAVA from PROLOG is not discussed. However, there are already several established approaches for querying JAVA from PROLOG. This is reasonable as the first approaches for the integration of PROLOG and JAVA came initially from the logic programming community. Two general approaches for object-oriented programming in PROLOG are discussed in [53, ?]. Another discussion, from a syntactical point of view, can be found in [104].

1.6 Supporting Publications

Preliminary results of our research on the integration of PROLOG and JAVA have been presented on international workshops and conferences. A selection of papers is given below.


2 Selected Programming Paradigms and Languages

With a history of almost 80 years, computers still are not able to understand natural language. Their level of understanding is restricted to the processing of binaries (machine code). As a result, we basically have two choices: we can tell a computer exactly what to do (the procedural approach) or we can provide behavioral rules together with a unified algorithm for the problem solving (the declarative approach). To instruct a computer we usually rely on higher, more abstract constructs which encapsulate low-level machine code and bridge the semantic gap. Of course, these constructs then have to be translated (compiled and/or interpreted) back to the machine-readable format. These constructs, and the way to combine them, form together what commonly is regarded as a programming language.

Starting from early programming languages such as FORTRAN which follows a strict procedural programming paradigm, new and more refined programming paradigms have been established over the course of time such as the logic or the object-oriented programming paradigm, with PROLOG and JAVA as popular representatives. Both programming languages excel in problem modeling and solving but have very different application areas. The JAVA programming language has been designed for the use in distributed environments such as the Internet, whereas PROLOG has contributed substantially in the field of artificial intelligence, knowledge engineering and deductive databases. Because modern software applications increasingly have to solve problems of very different domains, the cooperation of the different programming languages that are suitable for particular problem domains becomes more and more important. Therefore, a seamless integration of different programming languages and paradigms is desirable. In this thesis, we are especially interested in the integration of the logic programming paradigm and the the object-oriented programming paradigm, or rather the integration of PROLOG and JAVA.

Because some readers may only be familiar with one of these programming paradigms, we give short introductions to both programming paradigms and the programming languages PROLOG and JAVA. The following sections treat the relevant topics not exhaustively, they serve to give a broad overview on the used terms and concepts. More information can be found easily in well-established literature.
The following two Sections 2.1 and 2.2 characterize the logic programming paradigm and the logic programming language Prolog. We highlight important paradigmatic concepts and language relevant artifacts. Our remarks are loosely based on [10, 23, 32]. The Sections 2.3 and 2.4 characterize the object-oriented programming paradigm and the Java programming language, respectively. We loosely follow the remarks in [32, 43, 74]. Section 2.5 concludes the chapter with a discussion on the synergy of Prolog and Java.

2.1 The Logic Programming Paradigm

The concept of deduction as computation is nicely captured by the well-known slogan of Robert Kowalski who described the activity of programming simply by the equation \( \text{Algorithm} = \text{Logic} + \text{Control} \). Or in other words, any algorithm is the combined answer of the following two questions: 'what' characterizes a solution and 'how' it is found? Traditional imperative programming must answer both, logic programming only the first. In logic programming, the control is left to the abstract machine and the interpreter cunningly searches through the space of possible solution. The computation is based on a rule-based deductive resolution mechanism. The foundations of logic programming can be traced back to K. Gödel and J. Herbrand in the early 1930s. In fact, Herbrand already anticipated an early form of the main computational mechanism for logic programming languages that nowadays we know as unification. In the 1960s A. Robinson presented a first formal definition of unification and only ten years later, in the 1970s, formal automatic deduction has been recognized as a mechanism for computation.

In the following section, we will elaborate the basics that are necessary for an understanding of the more complex concepts of unification in Section 2.1.2 and the computational model for logic programming in Section 2.1.3.

2.1.1 The Language of First-Order Logic

Logic programs consist of special formulas, the so-called definite clauses, that have the syntax of first-order logic (or first-order predicate calculus). In first-order logic, symbols are used to 'predicate' an element’s properties. All elements are part of a fixed domain \( D \) of discourse. In logic of higher order, the arguments of a predicate can be, in addition to elements of \( D \) (first-order logic), sets or functions over \( D \).
(second-order logic) or even sets of functions over $\mathcal{D}$ (third-order logic). In the following, we restrict our discussion on first-order logic because primarily we are interested in logic programs. Languages of first-order logic have the following components.

1. An alphabet $\mathcal{A}$.
2. Terms defined over $\mathcal{A}$.
3. Well-formed formulas defined over $\mathcal{A}$.

Each component is defined as follows.

**Alphabet.** An alphabet is a finite (or at least countably infinite) set of symbols which usually can be disjunctively partitioned into a set of logical symbols and a set of non-logical symbols. The former is common to all first-order logic languages while the latter is domain-specific. Important examples for logical symbols are

- constants such as the propositional constants $true$ and $false$,
- the logical connectives ($\lor, \land, \neg, \rightarrow, \leftrightarrow$),
- a countably infinite set $V$ of variables ($X, Y, Z, \ldots$) and
- the quantifiers $\exists$ and $\forall$.

In addition, there are punctuation symbols such as brackets, commas, or colons. On the other hand, the non-logical symbols, also referred to as extra-logical symbols, are defined by ($\Sigma, \Pi$) where $\Sigma$ denotes the function signature, a set consisting of function symbols, and $\Pi$ denotes the predicate signature, a set of predicate symbols. Each function symbol and each predicate symbol is associated with an arity which is the number of arguments of the corresponding function or predicate. If the arity of a function $f$ is equal to zero, $f$ is said to be a constant. Function symbols represent functions, i.e. left-total right-unique relations, whereas predicate symbols are interpreted as relations. The usage of quantifiers distinguishes first-order logic from propositional logic.

**Terms.** The descriptive term, although commonly used in many contexts, is ambivalent. In first-order logic, a term is defined as follows.

**Definition 2.1.** (Term) Over a signature $\Sigma$ terms are defined inductively as follows:

1. A variable of the set $V$ of all variables is a term.
2. If $f \in \Sigma$ is function symbol with arity $n$ and $t_1, \ldots, t_n$ denote terms, then $f(t_1, \ldots, t_n)$ is a term.
We denote the set of all terms over the signature $\Sigma$ with $\mathcal{T}$.

The particular case $n = 0$ implies that constants are terms, too. Terms without variables as arguments are referred to as ground terms.

**Formulas.** Properties of terms are domain-specific. Predicate symbols allow for expressing semantics by constructing complex formulas from atomic building blocks.

**Definition 2.2.** (Formula) Well-formed formulas over a signature with terms $(\Sigma, \Pi)$ are defined as follows:

1. True and false are formulas.
2. If $t_1, \ldots, t_n$ are terms over $\Sigma$ and $p \in \Pi$ is a predicate symbol of arity $n$, then $p(t_1, \ldots, t_n)$.
3. If $F$ and $G$ are formulas, then $F \lor G$, $F \land G$, $\neg F$, $F \to G$ and $F \leftrightarrow G$ are formulas.
4. If $F$ is a formula and $X \in \mathcal{V}$ is a variable, then $\exists X. F$ and $\forall X. F$ are formulas.

For first-order logic, the simplest well-formed formulas are predicate symbols together with their arguments. Each argument is a term. These formulas are called atomic formulas. More complex formulas can be built from atomic formulas with the help of logical connectives.

**Logic Programs.** A finite disjunction of literals, i.e. atomic formulas or their negation, is said to be a clause. A clause with at most one positive literal is called Horn\textsuperscript{2} clause. For variables $X_1, \ldots, X_m \in \mathcal{V}$ and atomic formulas $A_1, \ldots, A_m$ and $B_1, \ldots, B_k$ a clause can be given as a formula of the form

$$\forall X_1, \ldots, X_m (A_1 \lor \cdots \lor A_m, \neg B_1 \lor \cdots \lor \neg B_k).$$

With the help of De Morgan’s law and the definition of implication, a clause easily can be rewritten into the following, equivalent, formula:

$$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_k.$$ 

In logic programming, we are interested in a particular class of clauses called definite clauses which compose logic programs.

\textsuperscript{2}Named after the American mathematician Alfred Horn (February 17, 1918 - April 16, 2001).
Definition 2.3. (Logic Program) Let $H, A_1, \ldots, A_n$ be atomic formulas. A formula of the form

$$H \leftarrow A_1 \land \cdots \land A_n.$$ 

is called definite clause. A finite set of definite clauses is called a logic program.

For $n = 0$, a definite clause is said to be a fact and a sole sequence of atomic formulas $A_1, \ldots, A_n$ is called query or goal. The precondition (right hand side of the of $\leftarrow$) of a clause is referred to as body while the conclusion (left hand side) as head. Therefore, a fact is after all a clause without body and a query is a clause without head. Variables in the body of a clause are called local variables.

2.1.2 Theory of Unification

Logic programs together with a basic inference rule allow for a unique computational model that is quite different from the object-oriented computational model. Logic programming is based on the substitution of logical variables.

Logical Variables. Logical variables are bound to terms over a given alphabet. However, the concept of a logical variable is different from the concept of a variable in object-oriented programming, see also Section 2.4.2. Notably, there are three main differences:

- Once the binding of a logical variable and a term is established, it can not be modified or destroyed any more. Therefore, subsequent assignments such as in imperative languages are not allowed. Although it is not possible to modify the binding of a variable to a term, it is possible to modify the current value of the variable. This contradiction, superficially considered, is explained in more detail by the next item.

- In contrast to variables in imperative languages, logical variables can be partially bound, i.e. the successive binding of logical variables leads to different values. For instance, if the variable $X$ is bound to the term $f(Y, Z)$ and successively the logical variable $Y$ is bound to the constant $a$ and $Z$ is bound to $g(W)$, then $X$ finally is bound to $f(a, g(W))$. In this way, the value of the variable $X$ varies during a computation until every occurring variable possibly is bound to a constant.

- Bindings of logical variables are bidirectional. For instance, if $X$ is bound to the term $f(Y)$ and later $X$ is bound to $f(a)$, $Y$ is bound to $a$. 
**Substitutions.** In the examples above we have used the term binding for the substitution of logical variables by terms.

**Definition 2.4.** (Substitution) A substitution is a total mapping \(\sigma : V \rightarrow T\) from variables to terms such that the number of variables that are not mapped to themselves is finite. Therefore, a substitution \(\sigma\) can be noted by

\[
\sigma = \{X_1/t_1, \ldots, X_n/t_n\}
\]

where \(X_1, \ldots, X_n\) are pairwise different variables, \(t_1, \ldots, t_n\) are terms and we assume that \(t_i\) is different from \(X_i\), for \(i = 1, \ldots, n\).

A pair \(X_i/t_i\) is called binding of the variable \(X_i\) to the term \(t_i\). If all the \(t_1, \ldots, t_n\) are ground terms, then \(\sigma\) is said to be a ground substitution. All variables actually replaced within a substitution \(\sigma\) are represented by the set \(\text{Domain}(\sigma) = \{X_1, \ldots, X_n\}\) and all variables that occur in a term of a binding are represented by the set \(\text{Codomain}(\sigma) = \{Y \mid Y \text{ is a variable in } t_i, \text{ for some } t_i, 1 \leq i \leq n\}\).

More generally, substitutions can be applied not only to terms but to more complex expressions such as literals, too. The composition \(\vartheta \sigma\) of the substitution \(\vartheta = \{X_1/t_1, \ldots, X_n/t_n\}\) and the substitution \(\sigma = \{Y_1/s_1, \ldots, Y_m/s_m\}\) is defined as the substitution that results from removing the pairs \(X_i/t_i\sigma\) from the set

\[
\{X_1/t_1\sigma, \ldots, X_n/t_n\sigma, Y_1/s_1, \ldots, Y_m/s_m\}
\]

such that \(X_i\) is equal to \(t_i\sigma\) and the pairs \(Y_i/s_i\) are equal to \(Y_i \in \{X_1, \ldots, X_n\}\). The composition of substitutions is associative and if \(\sigma, \theta\) are substitutions, \(E\) is a literal or a term, and \(E\sigma\) is the application of \(\sigma\) to the variables in the expression \(E\), then the equation

\[
E(\sigma\theta) = (E\sigma)\theta
\]

holds. A particular type of substitutions are those that only rename the occurring logical variables.

**Definition 2.5.** (Renaming) A substitution \(\rho\) is called renaming if its inverse substitution \(\rho^{-1}\) exists and is such that \(\rho\rho^{-1} = \rho^{-1}\rho = \epsilon\) holds with \(\epsilon\) denoting the empty substitution.

Substitutions are comparable. A substitution \(\sigma\) is said to be more general than the substitution \(\vartheta\), in signs \(\sigma \leq \vartheta\), if, and only if, there exists a substitution \(\gamma\) such that the equation

\[
\sigma\gamma = \vartheta
\]
holds. The binary relation $\leq$ over the set of all substitutions forms a preorder or quasiorder because neither anti-symmetry nor symmetry necessarily hold. Analogously, a term $t$ is said to be more general than a term $t'$, in signs $t \leq t'$, if, and only if, there exists a substitution $\sigma$ such that the equation

$$t\sigma = t'$$

holds. In this context the preorder is called variance and equivalence also can be stated alternatively as follows: $t$ and $t'$ are variants of each other if there exists a renaming $\rho$ such that $t$ is syntactically identical to $t'\rho$. The definition of the binary relation $\leq$ obviously can be extended to other expressions, for instance literals.

Given a substitution $\sigma$, if $W$ is a subset of $\text{Domain}(\sigma)$, then the substitution

$$\{Y/t | Y \in W \text{ and } Y/t \in \sigma\}$$

is called the restriction of $\sigma$ to the variables in $W$. Therefore, in contrast to the concept of variables in the object-oriented paradigm, a successive application of substitutions to terms leads to (possibly partially defined) values for the variables.

**Most General Unifier.** Now that we have defined the fundamentals, we can discuss a basic computational mechanism for the logic programming paradigm. Essentially this mechanism can be described by the solving of equations of the form $s \equiv t$, where $s$ and $t$ are terms and the binary predicate symbol $\equiv$ means syntactic equality over the set of all ground terms, which commonly is referred to as Herbrand universe\(^3\).

Because of the ambiguity of the non-logical symbol $=$ which might be misinterpreted as assignment as used for object-oriented programming, we have chosen to use $\equiv$ instead of $=$ throughout our subsequent descriptions. In addition, we use the infix notation with $\equiv$ to improve the readability.

For instance, if we write $X \equiv 1$ we mean that the logical variable $X$ shall be bound to the constant $1$. The $\equiv$ relation is symmetric, i.e. $X \equiv 1$ is equivalent to $1 \equiv X$. Note that the binding is bidirectional, $X$ can be bound to $1$, and vice versa. This affects the computational model considerably as it allows for a bidirectional parameter-passing mechanism, where input and output can switch places. More details on this important aspect of logic programming follow in Section 2.2. The substitution $\{X/1\}$ is a valid solution for our small example above. Already for this small example there are infinite substitutions that produce a valid solution. However, there are also examples for equations without any solution, for instance $f(X) \equiv f(g(X))$, unless we permit infinite terms.

---

\(^3\)Named after the French mathematician Jacques Herbrand (12 February 1908 - 27 July 1931).
Because substitutions are partially ordered by $\leq$, we can define a most general substitution, up to equivalence, that is valid solution. A substitutions that is valid solutions is said to be a unifier and the most general of all unifiers is called the most general unifier.

**Definition 2.6. (Most General Unifier)** Given a set $E = \{s_1 \equiv t_1, \ldots, s_n \equiv t_n\}$ of equations, where $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ are terms, then the substitution $\sigma$ is called a unifier for $E$ if $s_i\sigma$ and $t_i\sigma$ are syntactically identical, for $1 \leq i \leq n$. If $\sigma$ is more general than any other unifier of $E$, $\sigma$ is said to be the Most General Unifier (MGU) and for every other unifier $\vartheta$ of $E$ there is a substitution $\tau$ such that the equation

$$\sigma\tau = \vartheta$$

holds.

The definition above only refers to terms but, of course, it can be naturally extended to other, more complex expressions such as literals.

John A. Robinson\(^4\) made an important observation: the question if a set of equations of terms is unifiable, is decidable [80]. His proof provided an unification algorithm for a given set of equations that either returns a MGU or stops with a failure. A description of an efficient unification algorithm can be found in [51].

### 2.1.3 The Computational Model

The computational model for the logic programming paradigm differs in several points from object-oriented programming. Most notably are the following:

- Terms over a given signature $\Sigma$ are the only possible values.
- Logic programs are declarative but with a procedural reading.
- Computations are based on unification due to substitutions.
- There is no need for control structures.

In the following, we discuss properties of the logic programming paradigm. The characteristics of the logic programming language Prolog follow in Section 2.2.

---

\(^4\)John Alan Robinson was a British philosopher, mathematician, and computer scientist, born 1930 in Halifax (West Yorkshire). He contributed significantly to the foundations of automated theorem proving and also prepared the ground for the logic programming paradigm.
2.1 The Logic Programming Paradigm

Everything is term. In logic programming the fundamental building blocks are terms. The Herbrand universe is the set of all possible terms over a given signature. An ultimate consequence is that unlike in the object-oriented programming paradigm there is no type system present in the logic programming paradigm. Note that even the underlying alphabet is not fixed concerning the non-logical symbols. The interpretation of non-logical symbols may vary from program to program, too. In a logic program ‘+’ can denote, for instance, a concatenation of strings or alternatively the arithmetical operator. There are few non-logical symbols which are reserved and have a unique, predefined interpretation; e. g. the binary equality sign ‘=’, or rather ‘≡’ as we have defined it previously in Section 2.1.2, which denotes the syntactical equivalence over the Herbrand universe and, operationally, results in unification.

Declarative and Procedural Interpretation. Because logic programs are based on first-order logic and the special formulas called clauses, they have a declarative interpretation without any references to the computational process. However, to prove the head of a clause in a logic program all formulas in the body have to be proven first. This allows for a procedural interpretation, too. Therefore, it is possible to use the analogies of procedure calls and parameter passing for interpretation of logic programs, too. An elaborate example for a procedural interpretation of a logic program can be found in [32], Section 12.4.3. To evaluate a non-atomic goal there must be some sort of selection rule that determines the order how the atoms of the goal are selected for the evaluation. It can be shown that for pure logic programs the order of evaluation has no influence on the final results and thus a selection rule from the declarative point of view is not relevant.

SLD Resolution. Robert Kowalski proposed a refinement of resolution [46] which is called the Selective rule-driven Linear resolution for Definite clauses, or short SLD resolution. It became the basic inference rule for the logic programming paradigm.

**Definition 2.7. (SLD Resolution)** Let G be the goal B₁, ..., Bₖ and let C be the (definite) clause H ← A₁, ..., Aₙ. We say G’ is derived from G and C by the substitution σ or, equivalently G’ is a resolvent of G and C if, and only if, the following conditions are met:

- Bₘ, with 1 ≤ m ≤ k, is a selected atom from those in G.
- σ is the most greatest unifier of Bₘ and H.
- G’ is the goal (B₁, ..., Bₘ₋₁, A₁, ..., Aₙ, Bₘ₊₁, ..., Bₖ)σ.
Given a goal \( G \) and a logic program \( P \), an SLD derivation of \( P \cup G \) is a possibly infinite sequence of goals \( G_0, G_1, G_2, \ldots \), a sequence of renamed clauses \( C_1, C_2, \ldots \in P \) (to avoid variable capture) and a sequence \( \sigma_1, \sigma_2, \ldots \) of Mgu's such that \( G_0 = G \) and \( G_i \) is derived from \( G_{i-1} \) and \( C_i \) with help of \( \sigma_i \). Then, a finite SLD derivation of \( P \cup G \) is called SLD refutation if the last goal \( G_n \) in the finite sequence of goals \( G_0, G_1, G_2, \ldots, G_n \) is derived from \( G_{n-1} \) and the empty clause. The sequence \( \sigma_1, \sigma_2, \ldots \) of substitutions used in the refutation of \( P \cup G \) is called the computed answer substitution for the goal \( G \) in \( P \). The computed answer substitution is restricted to the variables that occur in \( G \).

It can be shown [22] that the SLD resolution is sound and complete with respect to first-order logic. If \( \sigma \) is the computed answer substitution for the goal \( G \) in \( P \), then \( G \sigma \) is a logical consequence of \( P \) (soundness). Furthermore, if \( G \sigma \) is a logical consequence of \( P \), then no matter which selection rule is applied, there exists a SLD refutation of \( P \cup G \) with a computed answer substitution \( \vartheta \) such that \( G \vartheta \) is more general than \( G \sigma \) (strong completeness).

SLD resolution implicitly traverses a search tree which represents alternative computations. The root node of the tree represents the initial query. Every node in the tree which is associated with a goal has for each resolvent obtained by SLD resolution an associated child node. A finite branch in the search tree that ends with a leaf node indicates that the initial query has been successfully solved if this leaf node is associated with the empty clause, otherwise it indicates a failure. Infinite branches, however, are possible. SLD resolution determines no search strategy for the traversal of the tree, e.g., depth-first or breadth-first search.

**Non-Determinism.** We already have discussed if the order how the atoms in a non-atomic goal are evaluated matters. The short answer was that the order does not influence the final result and as a result we can choose any selection rule for an actual implementation of the logic programming paradigm. However, if several clauses in a logical program define a predicate in a goal, a similar question arises. Which clause should be applied? There are simple examples where the selection of a suitable clause leads to termination or non-termination. This is where a special form of non-determinism comes into play, the 'don't know' non-determinism. We don’t know which clause leads to a successful computation. The theoretical model, as we have seen, is inherently non-deterministic. The result of the evaluation of a goal \( G \) in a logical program \( P \) is a set of substitutions obtained by the composition of all Mgu's that result from specific choices of clauses and that are restricted to the variables that appear in the goal \( G \). Of course non-determinism in the theoretical model must somehow be transformed to determinism, given that any implementation as programming language is evaluated on a physical machine which usually
works deterministic. How this issue is solved and which other consequences have to be faced when moving from the theoretical model to an actual implementation as programming language, is illustrated in the following section which introduces the general-purpose logic programming language PROLOG.

2.2 The Logic Programming Language Prolog

The first implementation of the logic programming paradigm was PROLOG which originates from programming in logic. It has been developed by Pierre Roussel and Allen Comerauer in 1972 based on Kowalski’s theoretical model and their own work based on automatic theorem proving and a formalism for natural language manipulation. Much later in the 1990s, the first ISO-Standard for PROLOG was introduced. Today, there is no major PROLOG implementation but many different. For efficiency reasons various predicates have been added as built-ins and also some control structures from imperative programming have been integrated. This clearly violates pure declarative programming and thus PROLOG as programming language has to be distinguished from the theoretical model. Nonetheless, PROLOG programs often shine with surpassing brevity, simplicity and clarity.

2.2.1 Characteristics

PROLOG’s basic syntax is almost the unmodified syntax of the language for first-order logic. Atomic building blocks in PROLOG are terms which are (inductively) defined as in first-order logic. A term is either a constant, a variable, or a compound of the form \( f(t_1, \ldots, t_n) \) with the n-ary function symbol (also referred to as functor) \( f \) and terms \( t_1, \ldots, t_n \). A constant in PROLOG is either an atom or a number. An atom in PROLOG is denoted in one of the following ways:

1. As a sequences consisting of letters, numbers and underscore ‘_’ that begins with a small letter.
2. As a sequence of special characters, for instance +, -, <, :, or ->.
3. As a sequence of any characters enclosed by single quotation marks.

A variable in PROLOG begins with a capital or underscore possibly followed by a sequence consisting of letters, numbers, and underscores. Special characters and single quotation marks are prohibited. The universal variable, which unifies with every term, is symbolized by a single underscore and is also referred to as ‘anonymous’ or ‘don’t care’ variable. A compound without variables is called ground term. In PROLOG, predicates define relations between terms. An atomic formula in PROLOG is
of the form \( p(t_1, \ldots, t_n) \) with the \( n \)-ary predicate symbol \( p \) and terms \( t_1, \ldots, t_n \). A predicate \( p \) with arity \( n \) is denoted by \( p/n \), for instance \( \text{member}/2 \). A more detailed notation for the signature of \( \text{member}/2 \) is \( \text{member}(\text{?Elem}, \text{?List}) \) which lists all arguments as variables with meaningful names. Our example defines the \text{member} relation between \( \text{Elem} \) and \( \text{List} \) that is true if \( \text{Elem} \) is a member of \( \text{List} \). The preceding question mark (\( ? \)) denotes for an argument that in query it either has to be \textit{bound} or \textit{unbound}, or, operationally said, be either an input or output parameter. Furthermore, any argument with a preceding \( + \) or \( - \) requires to be bound or unbound, respectively.

Complex formulas in \text{Prolog} are restricted to definite clauses. A \textit{definite clause} is a Horn clause with exactly one positive literal. Let \( h, a_1, \ldots, a_n \) be atomic formulas. A definite clause in \text{Prolog} is denoted as follows:

\[
    h : - a_1, \ldots, a_n.
\]

In comparison to the definition of Section 2.1.1 the conjunction symbols \( \wedge \) and \( \leftarrow \) have been substituted by comma and \( : - \), respectively. This has a practical reason: the symbols comma and \( : - \) can be easier communicated to a computer. Note that in \text{Prolog} the full stop at the end of a definite clause is part of the formulation. It is a necessary information for the \text{Prolog} interpreter that the end of a definite clause has been reached. From here on, in the context of \text{Prolog} and for brevity, we just say 'clause' but mean always definite clause.

Programs in \text{Prolog} essentially consist of \textit{facts} and \textit{rules}. A fact is a clause without body, i.e. a ground term with a full stop at the end. A rule in \text{Prolog} is a block of clauses that all have same head. A \textit{query} or \textit{goal} in \text{Prolog} is a clause with an empty head.

Given two goals \( g_1 \) and \( g_2 \). The \textit{disjunction} of \( g_1 \) and \( g_2 \) is denoted in \text{Prolog} by \( g_1 ; g_2 \) using the predefined binary operator \( ; \) with infix notation. Internally, \( g_1 ; g_2 \) will be resolved by two different clauses with identical head and the bodies \( g_1 \) and \( g_2 \), respectively.

For arithmetical purposes most \text{Prolog} systems offer several predefined operators and specific data structures:

- Integer and real number approximated by a floating point.
- Common arithmetic operators with an infix notation.
- Relational operators with an infix notation.
- The evaluation operator \textit{is} for arithmetic expressions.
2.2 The Logic Programming Language PROLOG

Arithmetic expressions with relational operators have to be based on ground terms for a successful evaluation. Moreover, arithmetical equality has to be distinguished from symbolic equality, i.e. symbolic equality of terms. The evaluation operator is usually has a variable as argument that unifies with the result of the arithmetical evaluation. For instance, the a call to ‘X is 1+2’ leads to the unification of the variable X with the value 3.

2.2.2 Important Language Concepts and Artifacts

PROLOG’s inference engine relies on a combination of the following mechanisms:

- **Unification** for a pattern matching with arbitrary complex data structures.
- **Resolution** for an automatic deduction of new knowledge.
- **Backtracking** as part of a depth-first search strategy for the solution space.

These mechanism altogether allow for very elegant programs in PROLOG.

**Unification.** In PROLOG unification is realized by an almost identical algorithm as compared to the one in [51] based on Herbrand’s thesis. The only major difference is the treatment of the occur-check which is a test for the occurrence of identical variables on both sides of an equation, for instance $X = f(X)$. Any valid substitution for such a variable leads to infinite terms, which usually are not supported. For reasons of efficiency, the time consuming occur-check is often omitted in most PROLOG systems. As a result, it is up to the programmer to avoid infinite terms.

In the theory of logic programming, the clauses of a logic program have no order. In PROLOG, however, a program is a sequences of clauses, i.e. the clauses are ordered. This is due to the fact that any PROLOG system has to run on a deterministic machine.

**Resolution.** In the execution of a goal a resolution step is applied in PROLOG. In PROLOG, resolution is based on SLD resolution as described previously in Section 2.1.3. The applied selection rule is defined as follows:

1. Given a non-atomic goal $G$ consisting of atoms $A_1, \ldots, A_n$ where $A_i$, $1 \leq i \leq n$, either denotes an equation or a predicate. If $G$ is called, then $A_1$ is evaluated first followed by the consecutive evaluation from left to right of all remaining $A_i$. 
2. If several clauses are applicable, the topmost clause in the associated logic program is selected first. If its evaluation fails, remaining applicable clauses are selected form top to bottom.

**Backtracking.** If the execution of a goal fails, PROLOG uses a chronological backtracking mechanism to recover a previous choice point. Bindings that result from the computation of the previous choice are destroyed during backtracking. If an alternative at the choice point exists, the computation is resumed and the next applicable clause is evaluated. If no alternatives, i.e. choice points, for the current goal are available, PROLOG tries to backtrack to an even previous choice point. If this procedure continues until no alternative choice points are left, PROLOG terminates with a failure. Backtracking is performed internally and thus remains completely hidden to the programmer. PROLOG traverses the search tree, implicitly defined by SLD resolution, depth-first. Naturally, there are constellation where the depth-first search is less efficient. However, this can often be avoided by a careful program design. PROLOG monitors and controls the search mechanism via its stack memory. Because all this happens internally, programs in PROLOG usually or more compact than a procedural reference implementation.

**Deterministic Aspects.** In order to create efficient programs, most PROLOG implementations offer procedural control structures. However, the trade-off for such extensions is that some portion of the declarative nature of logic programs is lost. Programs with procedural control structures are often more complicated to read.

In PROLOG, disjunction can be exploited for if-then-else constructs as known from imperative programming languages. Most PROLOG systems support the notation $C \rightarrow G_1; G_2$, which denotes that if condition $C$ is met then evaluate the goal $G_1$, or else the goal $G_2$. Its definition relies essentially on the cut operator, which is denoted by a single exclamation mark. The cut operator influences the backtracking mechanism in PROLOG. If a cut goal is called, all the previous choice points are deleted from the stack memory. In other words, all remaining branches from choice points previous to the cut goal are destroyed. Effective variable bindings are not affected by the cut. Use cases for the cut can be categorized broadly into three scenarios:

1. Fix the selection of a clause.

2. Fail a goal immediately without searching for alternatives.

3. Preventing PROLOG from generating alternatives through backtracking.

The second use case is realized with the argument-free predicate `fail` which stops the execution for immediate backtracking. Other applications of the predicate `fail`
are, e. g., for loops. A `fail` after a clause forces Prolog to search for all alternatives. If no alternative is left, the call ultimately fails.

The usage of cuts in Prolog program can be further distinguished into two categories. A cuts that does not alter the meaning of a Prolog program is commonly referred to as green cut, otherwise it is called red cut. More precisely, if we remove a green cut from a Prolog program, we still get an equivalent program which is only potentially less efficient. If we remove a red cut from a Prolog program, the program is modified and we may even risk termination. Cuts usually are used for efficiency reasons, for instance, to prevent expensive backtracking. However, cuts make a Prolog program less declarative and thus more difficult to understand.

In Prolog the negation of a goal can be expressed with the meta-predicate `not/1`. Negation in Prolog is defined operationally as negation as failure and allows for non-monotonic reasoning. If the the evaluation of a goal `g` terminates with a failure then `not(g)` succeeds and if `g` succeeds, `not(g)` fails. Moreover, if the evaluation of `g` does not terminate, the evaluation of `not(g)` does not terminate as well. In Prolog, the the predicate `not/1` is defined with the help of the predicates `cut` and `fail`. It is based on the so-called Closed-World Assumption, i. e. all positive knowledge can be inferred or is given in the form of facts. Beyond that, everything else is simply assumed to be false. This model differs from the definition of negation in classical logic. From the lack of success of `g` a success of `not(g)` cannot be concluded logically.

Dynamic Global Memory. Prolog’s global memory can easily be modified with the help of meta-predicates which allow to add or remove a clause from the global memory, respectively. For instance, the meta-predicate `asserta/1` a given clause on top of all clause in the global memory. Clauses saved to the global memory can be accessed at any time during the evaluation of goals. This allows for elegant programs without parameter passing. Note, that manipulations of the global memory are not undone during backtracking. A single clauses of given kind can be removed from the global memory with the help of the predicate `retract/1` and the predicate `retractall/1` removes all facts or clauses for which their head unifies with the single argument of `retractall` in one go.

Lists. Collections or lists are essential data structures for every programming language and are used to represent sets, sequences, and even more complex structures, for instance trees. In Prolog, any term can be a member of a list and list can have be completely different terms as members. This, in particular, distinguishes
lists in Prolog from lists in Java, or collections in general. In Java, every member of a collection shares the same type, up to polymorphism. A Prolog list is recursively defined and is given in the form \([H|T]\) where \(H\) denotes the list’s head, i.e. the first element of the list, and \(T\) the tail, the Prolog list that contains all the remaining members. The empty list is symbolized by the atom \([]\). An alternative notation for lists is the sequence of members enclosed by square brackets, for instance \([t_1, t_2, t_3]\) is a list of the terms \(t_1\), \(t_2\), and \(t_3\). In Prolog, the members of a list always have a fixed order. In Java, sets, for instance, guarantee no specific order for their elements.

**Modules.** Predicates in Prolog can be collected in units for organization. These units are called *modules* and define public interfaces to a set of given predicates and operators. Modules are part of the ISO-standard in 1990 but, unfortunately, are not implemented uniformly in each Prolog system. Most existing Prolog implementations provide custom systems for modules. Without modules, all predicates are organized in a single namespace. As a result, any predicate can call any predicate and dependencies between programs are not obvious. Therefore, if a Prolog program is modified, it is hard to predict which other programs are affected from the changes. Modules can reference other modules in order to make dependencies explicit.

**Logic programming and Databases.** Relational aspects and global memory facilities make Prolog a suitable candidate for database applications. In contrast to relational databases based on relational algebra, Prolog offers all the tools that are necessary for database applications in the form of built-in predicates and even much more. For *deductive* database, the query language DATALOG emerged and became prominent in 1977 during a workshop organized by Jack Minker\(^5\), and still is omnipresent[52]. Basic DATALOG can be considered to be a syntactical subset of Prolog with some further restrictions:

1. Function symbols are not allowed in DATALOG; basic terms are either constants or variables.
2. Disjunction in the body of a clause is not allowed.
3. Variables that appear in the head of a clause require to appear in a non-arithmetic positive literal in the body of the same clause.

---

\(^5\)Jack Minker is an American computer scientist, born 1927 in Brooklyn (New York). He is a leading authority in artificial intelligence, deductive databases, logic programming and non-monotonic reasoning.
2.3 The Object-Oriented Programming Paradigm

4. Variables that appear in a negative literal in the body of a clause require to appear in some positive literals in the body of the same clause.

The last two restrictions come from its evaluation mechanism which is different from PROLOG. Instead of a top-down evaluation in PROLOG, DATALOG has a bottom-up evaluation mechanism for goals. In PROLOG, recursion may cause the non-termination of the evaluation of a goal. In DATALOG termination is always guaranteed. Moreover, in contrast to PROLOG, the order of the clauses in a DATALOG program is irrelevant. Therefore, DATALOG is considered to be truly declarative.

However, DATALOG is not Turing-complete\(^6\). It has been used predominantly for data-centric applications; e.g., database applications, data integration, or information extraction. Whether PROLOG or DATALOG, compared to the relational algebra, the expressiveness of both languages is much higher. For instance, queries that involve an unknown number of joins cannot be expressed in the relational algebra. Due to the presence of recursion, in PROLOG and as well in DATALOG more powerful queries can be expressed. Even the common query language SQL for relational databases did not support recursive queries right from the start. Even though recursive queries now have been introduced to SQL, they come in the form of extensions that feel not natural in SQL and are much harder to implement compared to PROLOG or DATALOG.

2.3 The Object-Oriented Programming Paradigm

For a long time computer programs essentially have only answered the question 'How do you do it?'. With the introduction of object-oriented programming, a paradigm change in classical programming occurred and the new question 'Who does it?' had arisen. Structuring and organization of software became more and more important, and with it the monolithic structure of difficult to maintain programs of the past such as in FORTRAN or C were split into smaller units of limited scope. As the paradigm’s name suggests, the concept of objects play the center role.

2.3.1 Objects and Classes

In the object-oriented programming paradigm objects represent either mental constructs or more commonly items or beings from the real-world. In object-oriented information systems, an object encapsulates associated data (attributes or member

\(^6\)Named after the British pioneering computer scientist Alan Mathison Turing (June, 23, 1912 - June, 7, 1954). A programming language is said to be Turing-complete if it allows to compute every function which can be computed on a Turing machine.
fields) and a set of possible operations (methods or functional fields). Thus, an object can be considered as a record where the attributes constitute the object’s properties. The methods grant access to the attributes and allow for communication with other objects. Access to attributes is a static task while invoking a method is coupled with a dynamic selection, see also Section 2.3.5. In summary, the following three aspects characterize an object:

- **State.** Every object at any time is in a state which is reflected by the current values of its attributes.

- **Behavior.** All actions available to an object are represented by methods. A method can be invoked by sending a *message*. Such a message has to contain the name of the addressed object and method, together with possible parametric data.

- **Identity.** State and behavior may change over the course of an object’s lifetime, but its identity has to stay unaffected. An object’s identity usually is a unique and static name, often referred to as *reference*.

Similar to blueprints in engineering or architecture, *classes* define models for the composition of similar objects. These models specify attributes and methods for an entire category (or class) of objects. In this context, *instances* of a class are similar objects that share attributes and methods but differ in concrete attribute values; e.g., two car instances that only vary in color.

To create an instance of a class two tasks must be fulfilled: memory allocation and initialization of the object in a proper state. For the latter, not a method but an operation called *constructor* is used. Some object-oriented programming languages permit the definition of different constructors to instantiate classes with a different initial state or with side effects. Multiple constructors may only differ in the parameters, not in the name.

A class can be seen as an identical construct that reduces the number of definitions necessary for similar objects. In this way, the size of object-oriented programs decreases massively because methods and attributes have only to be defined once for a class. Instances then only need to initialize in a proper state.

In the object-oriented programming paradigm, there is also a classless alternative approach that is based on *prototypes*. Instead of classes, the organizing principle is that of *delegation*. Essentially, delegation describes the propagation of a method call from one object to its parent object. For more information on delegation and prototypes, we refer the interested reader to [48].
2.3 The Object-Oriented Programming Paradigm

Packages are another concept to improve structure and modularity in object-oriented software development. A package is an organizational unit for a set of classes that are related and have a certain common scope. Packages can be further grouped to build up greater libraries or frameworks. Every package constitutes its own namespace. In different packages, this allows for classes with identical names (identifiers) but different structure.

2.3.2 Information Hiding and Encapsulation

Important concepts for data abstraction, not only in the object-oriented programming paradigm, are information hiding and encapsulation. An object’s state and behavior can be hidden from other objects and thus only be accessible by the object itself. Attributes and methods that are accessible from outside are considered as public, otherwise private. The public view on an object is often referred to as the object’s interface. Note, that this should not be confused with the eponymous Java type to define groups of related methods with empty bodies. More details about the interface Java type follow in Section 2.4.2. The interface encapsulates the object from other objects, and the separation between interface and implementation provides a clean cut between use and definition. For instance, a method’s implementation is also always hidden, only its signature, i.e. identity, possible parameters and return type, can be made visible to the outside. If an object (sender) wants to communicate with another object (receiver), it has to use the interface of the receiver. For this, a sender has to transmit a message which contains the name (identity) of the receiver, and the name and possible parameters of the addressed method. Messages are the only way for objects to interact with each other.

2.3.3 Types and Subtypes

In object-oriented programming languages, the set of objects that are instances of a class are also associated with a type. In typed object-oriented programming languages, type and class can be used synonymously. A class definition introduces at the same time a new type. In languages with a name-based equivalence for types, the subtype relation must be explicitly introduced in the class definition. In typeless object-oriented programming languages such as Smalltalk [36] the class-to-type association is only implicit.

A type naturally corresponds to the interface of that instances of a class share. A subtype of a type T can be defined as type S for which the corresponding interface subsumes the interface associated with T. In other words, every message that an object of type T understands must also be understood by all objects of subtype S.
As a consequence, an object of subtype \( S \) shares all attributes and methods of an object with type \( T \). But an object with subtype \( S \) may also understand more messages than an object with (super-)type \( T \). On the one hand, a subtype \( S \) of \( T \) can be seen as an extension or specialization of \( T \). On the other hand, an object of type \( S \) can, in a certain context, be seen as an object of type \( T \).

A class without instances but subclasses is called *abstract* class. Calling the constructor of a subclass leads to a chain of calls to constructors of all superclasses. If multiple constructors exist in a superclass, the problem of a proper selection has to be solved (by the programming language).

In summary, types and subtypes form a hierarchy of types, which allow to uniformly address objects of a given type or subtype. They form a relation of compatibility between classes and their interfaces.

### 2.3.4 Inheritance and Polymorphism

The propagation of implemented attributes and methods from a class to its subclass which is called *inheritance* is similar to subtyping, but works on the level of implementation. Attributes and methods of the superclass are inherited by the subclass. The subclass can simply fall back on implementations contained in the superclass. New definitions are not necessary which saves lines of code. Furthermore, because methods are only defined in single spot, maintainability is simplified significantly.

In summary, inheritance forms a relation of class implementations and improves the reuse of code.

An important aspect of inheritance is that only the signature of an inherited method is fixed, while the implementation of a method may vary. If a method’s implementation is redefined in a subclass, it is called *method overriding*. For the programmer, this means that methods can have identical signatures, thus their interfaces are identical. Only the addressed object in a message decides unambiguously the behavior of the receiver object. Triggering different behaviors by sending identical messages to different objects is referred to as *polymorphism*. Inherited attributes can be modified, too. An attribute’s type, though, may not be modified as this is part of the attribute’s interface. For attributes this is called *shadowing* rather than overriding.

There are languages that allow for *multiple* inheritance or only for *single (or simple)* inheritance. In the case of single inheritance, there is a single superclass from which all other classes inherit. The resulting hierarchy of classes forms a tree, where every subclass can only have a single superclass. In the case of multiple inheritance, the relation is more complex and forms an acyclic, directed graph. An example for a programming language with multiple inheritance is C++. Multiple inheritance
can cause name clashes if a subclass has two superclasses with identical method signatures. Therefore, multiple inheritance is controversially discussed from the conceptual point of view and the actual implementation.

2.3.5 Dynamic Method Dispatch

At first, combining the already introduced concepts leads to a conflict. If we iterate an array filled with references of type $T$ and call the shared method $m$, which implementation of $m$ is executed for each object in the array? Note that the objects may have, next to $T$, any subtype $S$ of $T$. If the method $m$ of the supertype $T$ executes for all objects, polymorphism is ignored. But the available, static information (names) are not sufficient to solve this ambiguous situation, and therefore the compiler cannot decide it. We have to differentiate between the static type of the reference and the actual type of the objects in the array. Because this information is only available during runtime, we need to derive the type of the objects dynamically. This essential mechanism in the object-oriented paradigm is called *dynamic method dispatch* and is used even if a method invokes another method in the same object.

2.4 The Object-Oriented Programming Language Java

The Java programming language is a general-purpose language that is strictly object-oriented. It has been designed for the use in distributed environments and allows for concurrent program execution, be it on multiple cores or shared threads on a single processor or physically separated processors. Java adopted several concepts of the programming languages C and C++ but has been organized rather differently. Certain aspects of C and C++ have been omitted and instead ideas from other languages have been included. Many design decisions have been made with working economy in mind. As C. A. R. Hoare suggested in his classic paper on language design [38] that Java is not a language for research but for production.

2.4.1 Characteristics

The language design has avoided the inclusion of new or untested features. To further ensure the idea of a *safe* language, Java is strongly and statically typed. In addition, it does not allow any unsafe constructs, for instance access to arrays without

---

7 Sir Charles Antony Richard Hoare is a British computer scientist, born 11 January 1934 in Colombo, Sri Lanka. He has become famous for his sorting algorithm (Quick Sort) and a formal system for reasoning about the correctness of computer programs (Hoare logic).
preliminary index check. This has been avoided because it would cause a program to behave in an unexpected manner.

The Java programming language is a compiled language where programs are first translated to machine-independent byte code. Thereby, a program’s source files (with extension .java) are compiled into class files (with extension .class). The compiled code, then, is interpreted by the so-called Java Virtual Machine (JVM), a stack-based machine which accommodates the byte code to individual computer platforms. Programming errors can often be detected at compile time. However, there are also errors which can only be detected at runtime, for instance incorrect type castings. At runtime, all necessary classes are loaded and linked for the program execution. The JVM includes an optional just-in-time compiler which optimizes dynamically the program execution by using, e.g., advanced indexing techniques.

Java as compared to C is a programming language that operates on a higher level. Low-level operations from within the language such as access to the registers of the abstract machine or pointers to locations in the memory are not possible but Java fully supports references. Java has a built-in storage management, the garbage collector, and thus works without deallocation, such as in C which has proven to be a great source for errors. The Java Language Specifications are defined in [43] and the resulting instruction set and binary format are defined in the Java Virtual Machine Specification [105].

The huge success of the Java programming language certainly stems from the portability of its applications. The Oracle Corporation claims that Java runs on over 15 billion devices worldwide [60], ranging from Blu-ray players, over desktops, mobile and TV devices, and many more. Examples for application areas are web services, server-side applications, applications for mobile phones, micro processors, remote processors, sensors, wireless modules and various applications in electronic devices. Among other aspects, the separation of computation from the visualization by graphical user interfaces (GUI), reusability [33] and robustness have contributed substantially to Java’s success. Robustness in this context essentially means that the JVM automatically ensures integrity for every object involved.

2.4.2 Important Language Concepts and Artifacts

Because Java is strongly and statically typed, the concepts of class and type coincide, i.e., every class definition introduces a new type. Equivalence for types is name-based. As a consequence, the subtype relation must be explicitly introduced in the class definition which is done with the keyword extends. Java has single inheritance and its maximal type is java.lang.Object, the principles of delegation
and prototypes are not supported. Objects in Java are created with the constructor (method) of their type using the `new` keyword. In Java, a `block` is a group of zero or more statements between balanced braces. A method in Java consists of a `signature` and a `body`. The signature describes a method’s accessibility, return type, name and arguments. A method’s body is a block that contains the actual implementation of the method. To introduce common types that have to be specialized by subtypes for a concrete implementation, Java offers the concept of `abstract classes`. Abstract classes are marked by the keyword `abstract` and instead of normal methods they only contain `abstract methods`. Abstract methods are methods without body and are marked by the `abstract` keyword, too.

Classes in Java can be defined within another class definition. These classes are then called `inner classes`. Inner classes know all methods and attributes of the incorporating class, even private ones. Classes defined within a block are called `local classes`. Local classes often appear within a method’s implementation and have access to local parameters. `Anonymous classes` are local classes that have no name and no constructors. As a consequence, they do not introduce a type and can not be instantiated.

Java is case sensitive and there are several programming conventions to improve the readability of programs, for instance to capitalize class names. In order to reuse names in larger programs, there are namespaces in Java. Every class states its own `namespace`, the same is true for methods and program blocks, i.e., code in curly brackets. Namespaces form a hierarchical order which the compiler evaluates bottom-up and that allow the reuse of designators to overwrite definitions. Furthermore, classes can be organized in `packages`. Packages introduce their own namespace but have no hierarchical order. Therefore, every package must be named unambiguously.

`Variables` in Java are `modifiable`, which distinguishes them from `logical variables`, see also Section 2.1.2. Instead of a container or location (in the memory), variables in Java follow a `reference model` where not values but references for values are assigned which are typically stored in the heap. A variable `declaration` in Java is composed of a type statement followed by the name of the declaration. Every declaration in Java has a default value if none is explicitly assigned. Objects have a special default value represented by the keyword `null`. Declaration and value assignment often coincide.

Java mainly knows three different kinds of data objects that allocate memory: objects, variables and constants. A data object consists of a name, a values margin, an associated type and an actual value. Together with a list of valid operators, a data object forms a data type. In Java, there is a series of `standard data types`,
the so-called primitive Java types or short primitives: a logical type (boolean), integral types (byte, int, long, short), floating point types (double, float), and a character type (char). The names of primitive Java types are reserved lowercase keywords. Fields of primitive type always have an assigned default value of usually zero. In the case of boolean, it is false. Primitives are built into the language, and because they have no class definition, they can not be subtyped. They are not instantiated by using the new keyword. Instead, literals simply are assigned.

As data objects, arrays play a special role. They are containers that store ordered collections of either primitives or object references. All elements in an array share the same type, a mix of types is prohibited. The elements in an array are consecutively numbered with the first index equals zero. An array is consecutively declared by naming the type of its elements followed by opened and closed squared brackets and finally the array’s name. A single element is accessible in $O(1)$ just by referencing the element’s index. What is special about arrays is that they share properties of objects and primitives at the same time. Arrays have a constructor method to initialize an array with a given size and type for its elements and they extend the class java.lang.Object. Multiple references to entries in the array are also possible. On the other hand, arrays share with primitives that they have no class definition and can not be subtyped.

For strings of characters Java provides the utility class java.lang.String. The specialty of this class is that instances are immutable objects that can be initialized either by calling a constructor method or by directly assigning a string literal without the new keyword such as for primitive Java types.

Next to classes, Java offers interfaces to introduce new types. They are collections of methods and constants. An interface in Java is quite similar to a class. It defines a reference type and exposes methods for the interaction with the corresponding type. The name of the introduced type and the interface coincide. Like for a class, the source file of an interface is compiled into a class file. In contrast to classes, interfaces remain purely descriptive; they just define the communication layer and leave the exact behavior open. In this way, interfaces in Java accord to the abstract concept of interfaces as stated in Section 2.3.2. They only contain abstract methods, i. e., methods without implementation. The only exceptions are default methods and static methods. Default methods implement a default behavior and static methods are only related to the type and not to any of its instances. In addition, interfaces contain no fields, except those that are both static and final. Because they have no constructors, interfaces cannot be instantiated. Instead of extending an interface, a class implements an interface by using the same keyword followed by the interface’s name in the class declaration. Classes may imple-
ment several interfaces. Usually, a class that implements an interface has to declare all methods of the interface public. The only exception are abstract classes which declare all implemented methods abstract. In this way, a JAVA interface can be considered as a generalized JAVA type with limited behavior.

The interested reader may ask why JAVA offers interfaces as additional constructs next to classes. One reason is that interfaces can extend multiple other interfaces. However, this cannot be considered multiple inheritance because a method is not allowed to be defined in multiple, different super-interfaces. Nevertheless, interfaces compensate the limitations of JAVA regarding multiple inheritance a little bit and allow for more refined practical applications. Another reason is that interfaces let the programmer simply define communication contracts for types. Classes that implement an interface ensure support for the methods as defined in the interface. In so doing a safe communication between objects does not depend on their actual implementations but on their interfaces.

Methods in JAVA do not depend on the types of their arguments as stated in their signature but on the types of the actual objects that are arguments. This is called dynamic binding and distinguishes JAVA form classical programming languages such as PASCAL. Every subtype of an arguments type is valid type, too. Clearly this is a form of polymorphism but it is not completely general because its is limited to subtypes. For an explicit subtype-polymorphism that is more powerful, e. g., to express that the return type of a method depends on an argument type, the JAVA programming language was extended in version 5 by generic types or short generics. A generic type is a formal type parameter and is denoted by surrounding angle brackets. Generics allow for parametric classes and methods. Primitives are allowed as type parameter. A type parameter can be limited by supertype with the help of the keyword extends. Then, only subtypes of the supertype are valid type instances of the type parameter. Generics do not survive the compiler. Once the compiler has inferred all types and checked them, it applies a type erasure step for all generics and replaces them by their maximal supertype. Therefore, generics are basically syntactical sugar but improve the reusability of code significantly. However, critics claim that they are a main source for compatibility problems with older JAVA versions.

2.5 Synergy of Prolog and Java

In this section, we discuss the synergy of PROLOG and JAVA. We begin with a summary of intriguing aspects, flaws, and typical application areas of PROLOG in Section 2.5.1 and JAVA in Section 2.5.2. The different features of both programming
languages lead to a discussion on the synergy effects of an integration of PROLOG and JAVA in Section 2.5.3.

2.5.1 Advantages and Disadvantages of Prolog

'Prolog itself is perhaps the most beautiful, simple, yet powerful programming language ever created by man' [49], p. 173.

In the following, we summarize some intriguing features of PROLOG:

1. PROLOG is declarative. Programs can be encoded directly in the form of clauses in first-order logic and queries can be asked without bothering how the answer is internally computed.

2. PROLOG is logical. Programs in PROLOG have a logical reading and their formal correctness can be verified much more easily compared to programs in imperative languages.

3. PROLOG is relational. In contrast to functions, arguments of predicates in PROLOG can act as input as well as output. Therefore, functionality can be encoded in a very compact format.

4. PROLOG is homoiconic and a multi-level language. Knowledge can be encoded explicitly by clauses or implicitly with the help of rule-based programming. Homoiconicity in PROLOG allows for simplified meta-programming and because PROLOG can modify its own code, even self-modifying, adaptive programs can be developed.

5. PROLOG has a unique computational model with strong internal mechanisms. It can fall back on a pattern matching with arbitrarily complex structures (unification), automatic knowledge deduction (resolution) and a solution search (depth-first search) with backtracking.

6. PROLOG is compact. Due to backtracking, standard control structures such as for or while of imperative programming languages are not necessary in PROLOG and thus allows for very compact and elegant programs. Complex algorithm in imperative programming languages can often be implemented in PROLOG with few lines of code.

7. PROLOG is rule-based. Rules are natural constructs in PROLOG as they are naturally represented by terms.
2.5 Synergy of Prolog and Java

However, there are also disadvantages. First of all, the control of backtracking in PROLOG is limited. Only the cut operator provides some control of backtracking but the resulting programs are less declarative and readable. Although the abstract machines for PROLOG have been optimized, the efficiency of particular PROLOG programs remains limited. Compared to JAVA, PROLOG cannot be regarded a safe language. PROLOG has static type system and no uniform module system to organize namespaces and the visibility of predicates. Although an ISO standard has been defined, most PROLOG systems vary in handling, scope (built-ins, libraries) and support. The diversity of PROLOG systems also has scattered the PROLOG community and reduced the portability of PROLOG programs. This probably is also a reason for the minimal commercial interest in PROLOG and because the industry is often a driving force for innovation and productivity, there are only few sophisticated development facilities available for PROLOG.

Common Application Areas. Originally PROLOG was developed for natural language processing. Important other application domains clearly are rules-heavy systems. PROLOG has successfully been used in several areas such as artificial intelligence, expert systems, pattern recognition, decision support, knowledge extraction [88], verification [6], knowledge engineering [66], and deductive database systems [87]. It has also been used in bioinformatics [56], in machine learning [70], and for program analysis [86]. PROLOG has also been used to solve planning and scheduling problems. A more recent application of PROLOG is in context-aware systems [58].

2.5.2 Advantages and Disadvantages of Java

The JAVA programming language has evolved and grown along with the internet and the the World Wide Web. Today, JAVA is an important mainstream programming language and a myriad of applications are powered by JAVA. Probably due to the development for the ANDROID platform, JAVA is listed again as the most popular programming language of the Tiobe index [96].

In the following, we outline some of JAVA’s most intriguing features:

1. JAVA is portable. Almost any platform runs the JVM and thus is able to execute the bytecode of compiled JAVA programs.

2. JAVA is dynamic. Dynamic method dispatch and dynamic class loading allow us to start a program without having all classes or methods at hand. Missing parts can be dynamically loaded from, e. g., a remote machine.
3. **Java** is concurrent. Programs can be executed concurrently via threads without system operations that are specific to the used computing machine.

4. **Java** is structured. The object-oriented paradigm with inheritance and packages allows for highly structured and organized programs.

5. **Java** is safe. Its type system combined with a three-tier type check avoids almost any type errors before runtime. Programming features that are considered unsafe such as pointer references are still not allowed in **Java** and an automated garbage collector takes care of memory recovery.

6. **Java** is free. The **Java** software development kit (JDK) and the JVM are freely available.

7. **Java** is standardized. Both, the **Java** language and the JVM, are standardized by the Oracle Corporation.

8. **Java** is well supported. The commercial interest and a huge community have led to sophisticated development environments, program libraries, and programming tools for **Java**.

However, there are also some known issues with **Java**. The price for the reliability and the abstraction in **Java** is efficiency. **Java** programs usually cannot compete with the performance of low-level implementations in, e.g., C or C++. Another issue is that the type system combined with the message concept for objects often leads to verbose programs in **Java**. In contrast to **Prolog**, **Java** is little declarative, the programming style clearly is procedural. Moreover, there are also problems such as the well-known circle-ellipse problem [9] which illustrate the limits of object-oriented programming in general. In addition, rules have no natural representation in **Java**. Moreover, from a logical point of view, even the formal correctness of a **Java** program can hardly be verified.

**Common Application Areas.** As the mainstream programming language, **Java** is used in numerous domains. Common examples for application areas are distributed environments with a demand for concurrent process execution, such as in client-server systems. In addition, major **Java** libraries support the development of sophisticated graphical user interfaces. In industrial automation, **Java** is used for, e.g., enterprise resource planning systems. **Java** is also heavily used in embedded systems and especially for mobile devices, for instance mobile data acquisition devices in logistics. **Java** can be regarded as industry standard and because it allows for modular programming, system design with modeling languages such as **UML**, projects can be efficiently and clearly realized. Therefore, **Java** is well suited for the programming in the large.
2.5 Synergy of Prolog and Java

2.5.3 Synergy Effects

Modern software systems have to handle an increasing number of problems in very different domains and since not every problem can be elegantly modeled and solved in every programming language, an increasingly number of different programming languages are involved for the implementation of highly specialized subsystems which have to work together. Such subsystems usually evolve continually and often independently from each other and as a result, software ecosystems such as enterprise resource planning systems have reached gigantic proportions.

The complexity of modern software systems is extremely affected by the interaction of the integrated subsystems which jointly have to solve requests of the global system. Object-oriented programming languages such as Java are well suited for the abstraction and organization of different subsystems via objects, processes, and threads. However, modern software systems also require to become more and more intelligent which is another considerable reason for their growing complexity. Flexibility, adaptiveness, reconfiguration, and reasoning are indispensable for the automation of complex and versatile real-world processes. Moreover, knowledge of domain experts has to be integrated as part of the application logic, preferably in a clear, concise, and declarative manner, without ambiguity and redundancy. All these aspects of (artificial) intelligence are excellently addressed by the very essence of logic programming. Logic programming languages such as Prolog naturally provide the facilities for reasoning, inference, and declarative abstractions which are valuable building blocks for intelligent software components.

From what we have illustrated so far, it is clear that the differences between Prolog and Java particularly suggest a synergy of both programming languages – the different features of Prolog and Java complement each other very well. Although Java is a mainstream programming language and much more popular than Prolog, it can highly benefit from Prolog’s success in, e.g., artificial intelligence, natural language processing, deductive databases, or expert systems for various domains such as finance, defense, telecommunications, law, medicine, agriculture, engineering, manufacturing, and education. Given the fact that Java has such a predominant position in industry and academics, the object-oriented perspective of an integration of Prolog and Java is of utter importance. The synergy of Prolog and Java also offers novel program designs such as object-oriented programs that rely on powerful interpreters, rules, grammars, deductive database facilities, reasoning, and backtracking in Prolog. Due to the compactness of Prolog programs, the outsourcing of suitable problems to Prolog also leads to software systems that are more clear and concise. In addition, using the programming language that is most suitable for a given problem, usually increases productivity and simplifies the
subsequent program maintenance. However, for a beneficial synergy of PROLOG and JAVA elaborate integration approaches are essential.
3 Integration Approaches for Prolog and Java

The large number of approaches that address the integration of logic programming and object-oriented programming from many different angles underline the importance of the topic. Because the Java programming language has evolved over the time, constantly incorporating new programming features, ever refined concepts for the integration of Prolog and Java have emerged, too.

The following selected related work has been roughly divided into general integration techniques. Section 3.1 starts with the translation of Prolog programs to Java. Section 3.2 discusses the embedding of Prolog into Java. In Section 3.3 we deal with the integration of non-embedded Prolog system. Section 3.4 outlines several alternative approaches which have not been described thus far. Section 3.5, finally, constitutes the motivation for this thesis. From a concluding discussion, we derive important criteria for the seamless integration of Prolog and Java from the object-oriented perspective.

3.1 Integration via Translation

This section discusses how an integration of Prolog and Java can be achieved with the help of a translation of Prolog programs to Java.

3.1.1 jProlog

A common approach for Prolog systems is to compile their code to the Warren Abstract Machine\(^8\) (WAM) [98]. The WAM consists of a memory architecture with an instruction set and offers an optimized execution of programs in Prolog. Important techniques of WAM to improve subsequent interpretation are first argument indexing, choice point and tail call optimization, and the recovery of memory on failure. The WAM provides an execution model for a Prolog program which has been adopted for a direct translation of Prolog programs to Java. Such a translator is jProlog [24], a first-generation Prolog to Java compiler.

---

\(^8\)Named after the British computer scientist David H. D. Warren who wrote the first compiler for Prolog. The WAM has became a standard basis for many Prolog implementations.
Description. The jPROLOG approach uses the propagated execution model from WAM in combination with a binarization transformation [95] for the translation of PROLOG programs to JAVA. The authors of jPROLOG claimed that one goal of their approach was to learn JAVA and that therefore the speed of execution was never their concern. The jPROLOG approach is implemented in JAVA and is based on a compilation process with a continuation passing style. Listing 3.1 shows a small PROLOG program.

```
1 steam :- fire, water.
2 fire.
```

Listing 3.1: A simple PROLOG program for the translation with PROLOG CAFE.

During the binarization transformation every clause is transformed into a binary clause. A binary clause is a clause which has only a single atom in the body. Listing 3.2 shows the transformed example of Listing 3.1.

```
1 steam(Cont) :- fire(water(Cont)).
2 fire(Cont) :- call(Cont).
```

Listing 3.2: The transformed binary clauses.

The jPROLOG approach translates each binary clause into a single class in JAVA and every continuation goal is translated into a term object in JAVA which is registered for the execution in a hash table. If a clause is called, all goals in the body require to be initialized first at execution time in JAVA, even if the evaluation of the first goal will fail. This naturally leads to a worse performance.

The discussion of the jPROLOG approach, or rather the translation of PROLOG programs to JAVA in general, is postponed to the following Section 3.1.2 which describes at first a further development of the jPROLOG approach.

3.1.2 Prolog Cafe

The translator PROLOG CAFE originates from a translator (LLPj) which has been previously published by the authors of PROLOG CAFE to translate programs in the linear logic programming language LLP [4] to JAVA. The translator PROLOG CAFE is a further development of the just described translator jPROLOG [24] and improves it in several areas. For instance, PROLOG CAFE produces a smaller sized output in JAVA, supports parallelism, and has a better performance. In addition, PROLOG CAFE improves the interoperability of PROLOG with JAVA by particular predicates in PROLOG.
3.1 Integration via Translation

**Description.** The translator PROLOG CAFE generates for every predicate a set of classes that contains a single class as entry point and additional classes for the (possibly) different clauses as well as for every choice instruction. PROLOG CAFE avoids the execution overhead of the jPROLOG approach by using an improved translation method based on PROLOG’s box control flow model, see also Chapter 8 of [23]. With PROLOG CAFE, every continuation goal is translated into a single predicate object. In addition, memory overflow is omitted by executing translated code within a supervisor function. The PROLOG program from Listing 3.1 is translated by PROLOG CAFE to the Java classes as shown by Listing 3.3.

```java
import jp.ac.kobe_u.cs.prolog.lang.*;
public class PRED_steam_0 extends Predicate {
    public PRED_steam_0(Predicate cont) {
        this.cont = cont; // get Cont
    }
    public Predicate exec(Prolog engine){ // steam(Cont) :- fire(water(Cont)).
        engine.setB0();
        Predicate steam1 = new PRED_water_0(cont); // create water(Cont)
        return new PRED_fire_0(steam1); // call fire/0
    }
}
public class PRED_fire_0 extends Predicate {
    public PRED_fire_0(Predicate cont){ // get Cont
        this.cont = cont;
    }
    public Predicate exec(Prolog engine){ // fire(Cont) :- call(Cont).
        return cont; // call Cont
    }
}
```

**Listing 3.3:** The translated PROLOG program in JAVA.

Moreover, to improve the interaction with JAVA from the PROLOG side, the PROLOG CAFE approach provides custom predicates for the initialization of objects, the field access, and the invocation of methods.

- `java_constructor(+Class, ?Object)`
- `java_method(+Object, +Method, ?Return)`
- `java_get_field(+Object, +Field, ?Value)`
- `java_set_field(+Object, +Field, +Value)`
The Class argument is either atomic and specifying a fully qualified class name, or a compound term with functor denoting the constructor name and the arguments corresponding to the constructor’s parameters. If Object in java_method/3 is atomic, the referenced method in JAVA has to be static. Listing 3.4 illustrates the usage of the custom predicates with a simple example. A call of main/0 creates in JAVA a Person instance with the name Bob Andrews and which is not married.

```prolog
main :-
  java_constructor('company.Employee', E),
  java_method(E, setName('Bob','Andrews'), _),
  java_get_field('java.lang.Boolean', 'FALSE', False),
  java_method(E, setMaritalStatus(False), _).
```

Listing 3.4: Calling Java from PROLOG with PROLOG CAFE.

**Discussion.** Both approaches, jPROLOG and PROLOG CAFE, are implemented in JAVA and thus they are portable to any platform which runs the JVM. The PROLOG CAFE approach supports parallelism in PROLOG simply via threads in JAVA. Each thread represents a single PROLOG engine and is controlled from JAVA by the shared object (PrologControl) which can also be accessed from PROLOG. In addition, PROLOG CAFE produces more compact source code in JAVA than jPROLOG and supports the translation of modules in PROLOG to packages in JAVA. Because modules are not realized in all PROLOG systems uniformly, their translation almost always requires manual adaptions. The PROLOG CAFE approach also performs better [5] than jPROLOG.

However, compared to the performance of programs in SWI-PROLOG, the programs translated with PROLOG CAFE have been 3 to 10 times slower. Another issue is that a translated PROLOG program adds a significant layer to any object-oriented program. Moreover, the look and feel of the translated is not natural for JAVA developers. In addition, the execution semantics of PROLOG often bypass the optimization potentials of the JVM. Although further improvements already have been published [28] regarding the performance and the readability of the translated source code towards idiomatic JAVA, the performance gap to PROLOG implementations in C or C++ still has not been closed in every respect. The translation of PROLOG programs to JAVA is a static approach which relies on a static translation mechanism. Therefore, the representation of PROLOG predicates in JAVA is not customizable at all. Perhaps the most serious issue of the translation of PROLOG programs to JAVA is that built-in predicates of any given PROLOG system cannot be translated at all if their implementation is not written in PROLOG. The same is true for PROLOG programs that exploit extensions to PROLOG or involve native source code. This
confines the applicability of a straightforward translation of PROLOG programs to JAVA significantly.

### 3.2 Integration via Embedding

An alternative to the translation of PROLOG programs to JAVA is the embedding of PROLOG or logic programming aspects into JAVA.

However, along with the evolution of the JAVA programming language, new opportunities arise for novel integration approaches of PROLOG and JAVA. For instance, important new features introduced with JAVA 5.0 are reflection, generic types, and advanced meta-programming facilities based on annotations.

Reflection in JAVA [30] allows to inspect language elements such as classes, interfaces, fields, or methods at runtime. Their main purpose is to alter the behavior of JAVA applications at runtime. Reflection features extensibility. External classes can be accessed and instantiated simply calling their fully-qualified names. JAVA’s Reflection API [63] also offers methods to access and manipulate fields, methods and constructors in classes. Common examples for the usage of reflection are in visual debuggers and development environments. There are some known issues with reflection such as performance overhead due to non-applicable JVM optimization, security issues with security managers, and unintended exposure of internals.

Generic methods and classes can be used to implement more generalized source code; e. g., generalized collections [57]. Generic types are introduced with the help of type parameters that act as wildcards. In conjunction with the type inferring mechanism, generics provide type safety at compile time. However, the generic type information is not kept during compilation which is called type erasure. For instance the generic List<Person> type is simply compiled to List<Object>. The Person type parameter is not available at runtime. This is also the reason why generic exceptions are not possible.

An annotation is a form of metadata which decorates the source code of a program. Annotations usually provide information which do not affect the runtime behavior directly. However, they can be exploited at compile time or runtime, e. g., for a stronger type checking, error detection, or the generation of new source code. Annotation types are defined similarly to interfaces in JAVA. The annotation type name is preceded by a single @ symbol and the definition also may contain annotation type elements with optional default values. Annotation type elements are limited to primitive types, String, Class, enum, other annotations, and arrays of any type we have just mentioned. Before JAVA 8, annotations could only be applied to declarations.
Since Java 8 any type usage can be annotated. Common examples for predefined annotations in Java are @Override, @Deprecated or @SuppressWarnings. Annotations can have annotations, too. Such annotations are called meta-annotations.

3.2.1 tuProlog and P@J

The approach described in [21] is a framework called P@J which elegantly exploits generics and annotations for the integration of PROLOG and JAVA. The P@J framework provides a generic API layer for the modeling of PROLOG terms and a custom annotation layer for PROLOG-based extensions in JAVA. The framework is implemented for the use with tuProlog [25], a prolog implementation written in JAVA.

Description. tuProlog [25] is an open-source Prolog implementation in Java. Design goals of tuProlog have been minimality and configurability. tuProlog provides communication libraries in Java which allow to access a PROLOG engine from JAVA, and vice versa. A tuProlog engine is simple JAVA object which can load PROLOG programs and answer queries.

However, the provided JAVA library for the interaction with PROLOG is based on recreation. In order to query PROLOG from JAVA complex representations for terms have to be implemented with the help of the subclasses of the Term class, similar to JPL. Therefore, a JAVA programmer has to interpret terms in an object-oriented manner. However, tuProlog offers no automated conversion from objects to terms. With tuProlog, the results of a query to PROLOG also have to be translated back into meaningful JAVA objects which is a laborious, manual task. tuProlog provides custom PROLOG predicates which can be used to call JAVA, for instance the predicate java_object/3 instantiates an object in JAVA.

The integration framework P@J as described in [20, 21] is built on top of tuProlog and aims for a tighter integration with improved access to tuProlog from JAVA. To achieve this, P@J uses a generic type system in JAVA for terms in tuProlog. A custom annotation layer allows to embedded PROLOG code within JAVA classes that implements particular abstract methods in JAVA.

A custom hierarchy of generic JAVA classes is used to wrap PROLOG terms and thus offers the advantage of type checking at compile time. The root element of the type hierarchy is the generic, abstract Term<X> class. The subclasses of Term represent PROLOG constants, lists, compounds, and variables. A recursive pattern ensures that all wildcard variables are subtypes of the generic Term<X> class. For tuProlog compatibility an internal bidirectional conversion mechanism exists for terms in P@J to terms in tuProlog. In addition, the Term<X> class offers two methods, toJava and
3.2 Integration via Embedding

The interaction with tuProlog remains hidden as there is an automatic conversion between plain tuProlog representations for PROLOG structures and their P@J counterparts. The solution of a query in tuProlog is saved to SolveInfo instance which is then mapped to an instance of generic Solution<G,S> class in P@J, where G and S correspond to the called goal and the retrieved solution, respectively.

The P@J framework offers a set of custom JAVA annotations that allows to embed plain PROLOG code within abstract JAVA classes and to establish fixed mappings between queries in PROLOG and methods in JAVA. As a result calling a goal in PROLOG is reduced to a single method invocation on the JAVA side. The @PrologClass annotation is used to store a String array that contains the PROLOG code, see Listing 3.6 which has been extracted from [20]. In this way, the PROLOG source can be saved within the abstract JAVA class. Then, a JAVA programmer can link the abstract methods of the annotated class via the @PrologMethod annotations to the predicates defined in the @PrologClass annotation. Such a link explicitly contains all the necessary information for the mapping, i.e. the functor and the order of the method parameters as arguments of the associated predicate. Moreover, the elements of the @PrologMethod annotation clarify the intended usage of the abstract method and enable compile time checking; e.g., the hasMultipleOutput element can be used to notify the programmer to expect multiple solutions. Because all type usages in JAVA now can be annotated, P@J uses annotations such as @INPUT or @OUTPUT to

Listing 3.5: Instantiation of a simple PROLOG goal wrapper with P@J.

```java
ArrayList<Integer> list1 = new ArrayList<>();
list1.add(1);
list1.add(2);
// converting list1 into a P@J wrapper for Prolog lists
List<Integer> list2 = new List<Integer>(list1);
// defining the type of P@J wrapper for a Prolog goal
CompCons<Integer,CompCons<List<Integer>,CmpNil>> goal;
// instantiating the goal wrapper
goal = Compound.make("member", new Term<?,[]>{2,list2});
```
deal with the relational character of predicates in PROLOG. However, the annotated abstract classes have to be initialized with the help of a factory class in JAVA.

```java
@PrologClass
classes = {
    "remove([X|Xs],X,Xs).",
    "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).",
    "permutation([],[]).",
    "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs),
    permutation(Zs,Ys)."
}
public abstract class PermutationUtility {

    @PrologMethod (link="remove($1,$2,$0)",
    style=PrologInvocationKind.FUNCTIONAL)
    abstract @GROUND List<Int> remove(@INPUT @GROUND List<Int> c1,
    @INPUT @GROUND Int i);

    @PrologMethod (link="permutation($1,$2)",
    multipleOutput=true,
    style=PrologInvocationKind.RELATIONAL)
    abstract @GROUND Iterable<Compound1<List<Int>>> perms(@HIDE @INPUT
    @GROUND List<Int> c1, @OUTPUT @GROUND Var<List<Int>> c2);
}
```

Listing 3.6: A PROLOG class in P@J.

The P@J framework also offers an object-to-term conversion for user-defined classes. Such classes have to be annotated with @Termifiable. The framework uses JAVA’s Introspection API to analyze the annotated class to infer a corresponding representation as PROLOG term. For instance, given a with @Termifiable annotated Person class with attributes for a person’s name (literal) and age (integer). The corresponding representation of a Person instance with the name Bob and age equals 12 as term is given by 'Person'(property('name', 'Bob'), property('age', 12)).

Once given a term, the framework searches for a class annotated with @Termifiable and for which the class name that is equal to the term’s functor. If such an class exists, an instance is created and the corresponding attribute values are set via reflection. The annotation layer of P@J can be further exploited with a custom annotation processor [61], a feature introduced in JAVA 6 which allows to easily extend the JAVA compiler. A registered annotation processor is instantiated by the compiler for the analysis of existing annotations in a sequence of rounds. In this way, the annotation processor can verify the compliance of annotated elements with the P@J framework. This increases the reliability of programs written with P@J.
3.2 Integration via Embedding

**Discussion.** The P@J approach shows nicely how to exploit new Java features such as generics or the annotation facilities that were introduced in previous versions of Java. Moreover, P@J verifies that the dynamic evolution of the Java enables novel approaches to the integration of Prolog and Java.

However, the P@J framework still leads to verbose expressions for Prolog structures in Java. Although the generics improve the reusability of source code considerably, the excessive usage of generics decreases the readability. Another problem is that generics do not work for primitive Java types. Therefore, for primitives the type information is lost during a conversion to tuProlog. The custom wrapper classes of P@J for the wrapper classes of primitive Java types may further complicate the development due to similar names, e.g., Boolean (Java) for Boolean (P@J). All classes have to be annotated for the conversion to tuProlog with P@J. This is a problem if the source code is not accessible, for instance, if the classes are only available as Jar. Moreover, the annotation layer of P@J has no full ISO compliance, there is no support for, e.g., DCGs, exceptions, and modules. P@J heavily relies on Java’s introspection API to analyze the annotations and dynamic proxy classes to initialize their abstract classes that are annotated for the use with tuProlog. Both techniques affect the runtime performance. Because there is no clean separation source-wise between Prolog and Java, an independent development in Prolog from Java is difficult. Finally, the conversion mechanism for Java objects and Prolog terms only applies to plain Java objects which are made of collections and primitive Java types.

3.2.2 Smalltalk and SOUL

Another interesting approach to achieve a seamless integration of different programming languages is based on the concept of linguistic symbiosis. A linguistic symbiosis of programming languages relies on a hidden and automated mechanism that manages the interaction of the involved programming languages. In this way, linguistic symbiosis enables different programming languages to call each other transparently and without explicit invocations. As a result, a replacement of program elements is possible: procedures, functions, or methods in one language can be exchanged by corresponding program elements of the other language, without major modifications of other program elements that are associated with the exchanged program elements. The term linguistic symbiosis was used for the first time in a work about an interpreter written in C++ [40]. Almost all components of this interpreter are replaceable by components written in the language for which the interpreter applies to.
The concepts of linguistic symbiosis also have been applied to achieve a symbiosis of logic and object-oriented programming [11, 37]. Moreover, the seamless integration of rule-based knowledge and object-oriented functionality based on linguistic symbiosis has been proposed in [26]. Instead of JAVA and PROLOG, the approaches described in [11, 37, 26] address the object-oriented programming language SMALLTALK and the embedded logic programming language SOUL.

**Description.** SMALLTALK is an object-oriented, dynamically typed, reflective programming language which has been publicly released in 1980 by Alan Kay and Dan Ingalls. The Smalltalk Open Unification Language (SOUL) is a PROLOG-like logic programming language built on top of SMALLTALK. In addition to backward chaining, SOUL features a production system with forward chaining.

SOUL is naturally embedded in SMALLTALK as it is implemented in SMALLTALK. Therefore SOUL can interact directly with objects in SMALLTALK which may also contain logic variables of SOUL. Although the representation of rules in SOUL is similar to PROLOG, their syntax differs; e.g., logical variables are denoted by a literal with a preceding question mark. Moreover, SOUL has no equivalent to PROLOG’s cut operator. Listing 3.7 shows a simple example of a rule in SOUL which proves if the age property of an object unified with ?p is at least 18.

```
?p is0fFullAge if
?p getAge = ?a & ?a >= 18
```

Listing 3.7: A simple rule in SOUL.

The approach described in [37] for the interaction of methods in SMALLTALK and predicates in SOUL provides an automatic dispatching mechanism which is based on a simple one-to-one correspondence of method names in SMALLTALK and predicate names in SOUL. If a SMALLTALK method is invoked without definition in SMALLTALK, the call is delegated to SOUL as predicate call. The name of the called predicate in SOUL is equal to the name of the method initially invoked in SMALLTALK. The receiver object in SMALLTALK is represented as the first variable of the call and remaining method parameters delegated to arguments of the called predicate. On the other side, if a predicate within a rule’s premise cannot be proven in SOUL, a message to an identically named SMALLTALK method is sent to the object which has been bound to the first variable of the rule’s premise. The remaining variables are bound to corresponding parameters of the invoked method. The invocation of a SMALLTALK method from SOUL with a non-boolean return is indicated with an equality sign. For this purpose the treatment of the equality sign is special in SOUL. A message’s answer from the left hand side of = is unified with the right hand side. If **getAge** in the premise of the rule in Listing 3.7, i.e. the part following
the if, is unknown to SOUL, a message is sent to the SMALLTALK object which has been bound to ?p. The result, then, is unified to ?a.

SOUL also supports forward chaining rules. They are triggered in SMALLTALK when the state of application relevant objects changes. All relevant objects which are considered facts are monitored. SOUL fires relevant rules automatically upon a changed state of a monitored object. The involved mechanisms are out of our scope and therefore omitted here.

Discussion. Even though the integration of SMALLTALK and SOUL differs in several parameters from a integration of PROLOG and JAVA, certain issues, however, remain the same regarding the paradigmatic distance between logic and object-oriented programming. Former work based on linguistic symbiosis often dealt with combining solely object-oriented languages and, thus, the mapping of messages. For the integration of a logic programming language and an object-oriented programming, the automation of the mapping of predicate calls to methods invocations, and vice versa, is remarkable and shows how automation can be used to hide complexity from the programmer.

Many interoperability problems that exist between PROLOG and JAVA have been bypassed by the very tight coupling of SOUL and SMALLTALK. However, this has been achieved by modifying the internals of SOUL and SMALLTALK significantly. For instance, the syntax of SOUL has adapted to the syntax of SMALLTALK, and the semantics of SMALLTALK have been extended to support unbound variables in SOUL. Such modifications usually are not possible in JAVA, at least not without modifying the JAVA programming language considerably which leads to significant portability issues. In the following section, we describe an approach that extends JAVA in order to integrate logic programming concepts.

3.2.3 Logic Java

Another option for the integration of logic programming and object-oriented programming is to alter the JAVA programming language. There are mainly two ways to achieve this:

- A custom compilation process.
- An extended JAVA Virtual Machine (JVM).

The following approach LOGIC JAVA [50] is based on the latter and has been designed to particularly simplify the solving of search problems in JAVA.
Description. The Logic Java approach preserves the syntax of Java and uses annotations to add concepts known from logic programming. Therefore, standard Java compilers can be used without modifications. However, the Logic Java approach replaces the usual Java virtual machine by an extended JVM called the Symbolic Java Virtual Machine (SJVM).

The SJVM is built on top of the conventional JVM and adds features such as logic variables, choice points, and backtracking, which are known from virtual machines for logic programming languages, e.g., the Warren Abstract Machine [98]. Without the supported logic features, the SJVM behaves just as the conventional JVM and only leads to little overhead.

Logic Java uses annotations to notify the SJVM to treat annotated elements particularly. In this way, logic computations can be merged with conventional computations in Java. If a single Java routine within a logic computation does not use any of the logic concepts, it behaves conventional. There is no need in Logic Java to annotate conventional Java routines as conventional. However, the outermost computation is always a conventional Java routine.

There are three common ways for introducing logic variables to Java:

- Type wrappers in Java for logical variables.
- Type annotations for logic variable.
- Naming conventions.

Type wrappers, see also Section 3.3.2, add additional object initialization for logical variables and the costs for (un-)boxing affect usually the runtime performance. Naming conventions often prove error-prone and are less flexible, see also Section 3.3.4. A flexible alternative are annotations which are also used by Logic Java to tag logical variables. The @LogicVariable annotation of Logic Java identifies primitive Java types as logical variables. The predefined, generic Solutions<T> class implements the java.lang.Iterable<T> interface and has a special role in Logic Java. It provides the interface between logic and conventional computations. Processing a logical computation the SJVM first collects all feasible solutions, removing empty solutions, and then returns an instance of Solutions<T>. Therefore, all queries with Logic Java can be considered as if wrapped by the findall predicate in Prolog. To avoid infinite computations Logic Java offers the optional @Search annotation to choose between depth-first search or iterative deepening.

The Listing 3.8 shows an example from [50] which illustrates the usage of Logic Java. The Fermat class can be used to calculate suitable positive integers $a, b, c,$ and $n$ that satisfy the equation $a^n + b^n = c^n$ of Fermat’s famous last theorem.
3.2 Integration via Embedding

```java
public class Fermat {
    @LogicVariable
    protected int a, b, c, n;

    @Search(strategy = SearchStrategy.ITERATIVE_DEEPENING,
            deepeningIncrement = 5)
    public Solutions<Integer[]> fermat(){
        if (power(a,n) + power(b,n) == power(c,n)){
            Integer[] solution = {a, b, c, n };
            return new Solutions<Integer[]>(
                new Solution<Integer[]>(solution));
        } else {
            return new Solutions<Integer[]>(new EmptySolution());
        }
    }
}
```

Listing 3.8: Solving Fermat’s Last Theorem with Logic Java.

Logic Java only provides a constraint solver and thus no non-trivial unification mechanism. In addition, most Prolog systems have complex mechanisms to effectively reduce the search space in order to improve their efficiency. However, unification can be considered as a special case of constraint solving. Therefore, unification in Logic Java can simply be simulated using equality constraints.

Internally, variables annotated with @LogicalVariable are handled by the SJVM as follows. If an object is instantiated by the new statement of Java, the JVM generates an internal representation of the object reference. It then checks whether fields of the underlying class have been marked with @LogicVariable. If so, it initializes the fields to logic variables. Otherwise, they simply take default values and are used as constants.

We only outline here how the SJVM deals with choice points and how backtracking is realized internally. Based on the bytecode of a class file, all choices, such as conditional jumps or switching instructions lead to the generation of choice points. Constraints are generated and describe the condition under which a considered branch can be entered. By adding constraints to the internal constraint store the solver checks for feasibility. If the current set of constraints is not solvable, the computation within the considered branch is abandoned and backtracking occurs. To handle different types of constraints efficiently, Logic Java has integrated multiple solvers.
3 Integration Approaches for Prolog and Java

Discussion. The logic extension LOGIC JAVA enables programmers to implement solutions for search problems such as combinatorial problems in scheduling. Because JAVA’s syntax is not altered and most mechanisms remain hidden from the programmer, it is very close to common JAVA and thus easy to learn.

Although suitable for formulating search problems in JAVA, the search for a solutions can be lengthy with LOGIC JAVA. The internal search mechanism always determines all solutions before returning an iterable instance of the Solutions class. Individual solutions cannot be obtained lazily, one after another. Another problem is that it is difficult to find an optimal solution. LOGIC JAVA provides no refined optimization algorithm. In addition, logical variables are limited to (primitive) class members. Local variable cannot be used as logical variables. The SJVM supports no concurrency and symbolic computations with LOGIC JAVA cannot be combined with threads.

Although LOGIC JAVA provides a solution search mechanism similar to PROLOG, i. e. depth first search with backtracking, it does not support the development of rule-based systems. Compared to LOGIC JAVA programs, PROLOG programs are still more clear and concise. Moreover, because there is no cut operator available in LOGIC JAVA, equivalent PROLOG programs can be implemented more efficiently.

For standard JAVA programs the SJVM performs not much worse than the conventional JVM. However, the reflection of @LogicalVariable annotations at runtime affects the performance. Finally, the most notable drawback of LOGIC JAVA is that it requires a modified JVM which considerably limits portability.

3.3 Integration via Communication Interfaces

Many popular PROLOG systems are not embedded into JAVA but stand-alone, for instance BPROLOG, the CIAO system, GNU-PROLOG, SWI-PROLOG, TU-PROLOG, the XSB system, or YAPROLOG. The integration of a PROLOG process and a JAVA runtime is realized by communication interfaces. In the following, we discuss several different approaches that provide custom communication interfaces for the integration of a non-embedded PROLOG systems and JAVA.

3.3.1 Interprolog

INTERPROLOG [15] is a bidirectional interface between JAVA and the XSB system [93, 94], which is logic programming system that subsumes PROLOG and offers a powerful deductive database engine. The implementation of the XSB system is also based on the WAM. The custom interface INTERPROLOG for XSB exploits JAVA’s
3.3 Integration via Communication Interfaces

serialization mechanism [59] in order to encode and decode JAVA objects to streams of bytes. Serialized JAVA objects are deserialized with the help of definite clause grammars [75] in PROLOG. Messages from PROLOG to JAVA are serialized in the same manner and can be deserialized in JAVA.

**Description.** To understand the approach of INTERPROLOG we explain shortly the serialization mechanism in JAVA. Object serialization produces a byte-sequence that conform to JAVA’s Object Serialization Stream Protocol [64]. From this sequence the state of an object at the time of serialization can be reconstructed, even on a different JAVA Virtual Machine (JVM). Instances of classes that implement the java.io.Serializable interface are valid candidates for serializing into a binary form. To correctly store an object all referenced objects must be valid candidates, too. Objects simply can be stored to and reconstructed from object streams by calling the streams writeObject, readObject methods. The stored information even enables remote object access, for instance from different JAVA virtual machines.

INTERPROLOG exploits the mechanism described above to produce its data exchange format, the binary form of an object. The binary form then is passed to the XSB system. Once a message from JAVA is passed to PROLOG, it is parsed with the help of definite clause grammars (DCGs) in PROLOG. DCGs have been proposed first by F. Pereira and D. H. Warren in [75] and are a natural extension of context-free grammars in predicate logic. The DCGs of INTERPROLOG implement JAVA’s Object Serialization Stream Protocol in order to derive from the stream of bytes of a serialized JAVA object a representation as PROLOG term. Listing 3.9 which has been extracted from [16] shows the derived raw term for a serialized java.lang.Integer instance with value 13.

```
object(
  class(java.lang.Integer,long(316842148,-142506184),
  classDescInfo([int(value)],
    class(java.lang.Number,long(-2035509987,194306187),
      classDescInfo([],2, null))",
    [] + [] + [13])).
```

Listing 3.9: A raw term in INTERPROLOG for an Integer instance.

For more complex objects, the size of the corresponding object reference trees grows, which leads to terms of considerable size in PROLOG, too. As characteristic for PROLOG, the DCG is not only used to parse the input byte-sequence but also to output a byte-sequence from a given object/2 term. The ipObjectSpec/4 predicate specifies the transformation and the javaMessage/6 predicate synchronously sends
messages to Java. Exceptions in Java are caught, otherwise PROLOG waits for the return. Both predicates are part of INTERPROLOG’s low-level PROLOG API.

More user friendly access to Java provides a high-level API which consists of the java predicates. Listing 3.10 illustrates the usage of the java/3 predicate. We invoke the substring method of the String class in Java to obtain a substring from a given string that begins with the character at the specified index.

```
?- java(string('Hello World'), string(SubString), substring(int(6))).
SubString = 'World'.
```

Listing 3.10: Usage of the java/3 predicate in INTERPROLOG.

For the queries with the java predicates INTERPROLOG relies on a simple, fixed conversion of basic PROLOG types to primitive Java types. The conversion mechanisms is based on type annotations in PROLOG such as byte(+Byte). For instance, an int array instance in Java which contains the three integers 1,2, and 3 is represented by the term array(int,[1,2,3]).

For queries from Java to PROLOG, INTERPROLOG only supports PROLOG goals in string format. Listing 3.11 shows how a socket-based communication to an XSB engine is initialized. The javaMessage/2 predicate is called in PROLOG and used to invoke a JAVA method in order to print the message 'Hello from Prolog, Java world!' to the console. Instead of a socket-based communication, the INTERPROLOG offers also the communication with an XSB process via JAVA’s Native Language Interface (JNI) [62]. In so doing, every ProcessEngine instance is associated with its own XSB process. The input parameter for the deterministicGoal method in Line 2 of Listing 3.11 shows the called PROLOG goal in string format. In PROLOG, the called javaMessage/2 predicate synchronously sends a print command to JAVA.

```
ProcessEngine engine = new XSBSubprocessEngine();
if (engine.deterministicGoal("javaMessage('java.lang.System'-out,println
(string('Prolog calls Java'))")
System.out.println("Java has received a call from Prolog!");
}
engine.shutdown();
```

Listing 3.11: Calling PROLOG from JAVA with INTERPROLOG.

The deterministicGoal method has several implementations in JAVA. In the example above, we see the version of deterministicGoal which is only applicable for goals without unbound variables and returns a boolean that indicates the success of a call to PROLOG. Another version of deterministicGoal returns an array that contains the bindings of the logical variables in the query to PROLOG. If the query
to Prolog fails, the deterministicGoal method returns null. Subsequently, Interprolog has included the support for calls to non-deterministic predicates in Prolog. Instead of returning only the first solution, the goal method returns immediately an Iterator instance which then can be used to traverse all possible solutions. In doing so each solution is lazily returned by Prolog due to backtracking. If the query fails in Prolog, the Iterator instance simply has no elements.

Discussion. Interprolog provides great facilities for calling Java from Prolog. But it lacks a high-level support for the inverted direction. Queries to Prolog from Java have to be created from strings, too. On the Prolog side, complex term structures have to be derived each time an object is transferred. For complex objects, the overhead and the size of resulting raw terms are considerable.

Because of the usage of a DCG, there is quite a portion of Prolog code that has to be ported in order to run Interprolog with other Prolog systems. Next to the XSB system, further support for Swi-prolog has been announced.

Relying on Java’s serialization mechanism has several drawbacks. On the Java side, all classes involved have to implement the Serializable interface and its members have to be serializable. Not only the properties of an serialized object are stored but the full class name and even the class definition, too. This leads to considerable overhead. Therefore, the serialization of different Java types performs worse than the serialization of several instances that share the same type. Moreover, a serializable subclass of a non-serializable superclass requires access to no-arg constructor of the superclass. Another often discussed issue of serialization is security. Making a class serializable grants to some extent public access to all class members, even to private class members. To avoid this, a class member can be declared transient in order to prevent its declared transient. Such considerations also affect the productivity.

3.3.2 JPL

Another interesting approach is JPL [89] which is also a bidirectional Java-Prolog interface just like Interprolog. It is shipped together with the free Prolog implementation Swi-Prolog [99]. JPL is based on Swi’s Foreign Language Interface (Fli) [100] and Java’s Native Interface (JNI) [62]. On the Java side, JPL consists of a library with classes representing atoms, terms, and logical variables and several methods for querying Prolog. On the Prolog side, JPL offers predicates to initialize objects and to execute Java methods from Prolog. Native C functions in Swi-Prolog work as glue code. The documentations of the latest APIs for JPL are publicly available [90, 91].
Description. JPL is based on JAVA’s JNI which enables JAVA to call or be called by foreign program libraries, for instance libraries written in C or C++. In addition, the JNI supports the interaction of JAVA with native applications, i. e. programs that are specific to hardware or operating system platforms. Official information about the JNI is publicly available [62]. SWI-PROLOG’s FLI is an interface to C that allows us to define foreign predicates as functions in C. The cooperation of the JNI and the FLI achieves a good performance. In JAVA, JPL has two levels of abstraction for the interaction with PROLOG. A high-level interface offers methods to query PROLOG. The query format is either based on plain strings or the goals of the query have to be constructed from specific JAVA types that represent PROLOG data structures such as atom, term or variable. An additional low-level interface offers raw access to PROLOG via the foreign functions of the FLI. Because the low-level interface is considerably less accessible, we omit further details.

Listing 3.12 illustrates the usage of JPL. First, we declare a logical variable X. Then we pass to the constructor of Query a Term instance that represents the PROLOG goal father(dietmar, X) which is subsequently used to query the father(+Name, +ChildName) predicate in PROLOG. In JPL, a binding of a logical variable in PROLOG due to unification is represented a key-value pair within a java.util.Hashtable. Because the Query class implements the java.util.Enumeration interface, all solutions can be traversed via the nextElement method. A query can be tested for one or more solution with the method hasMoreElements. For convenience and in order to avoid nasty type casts, the hasMoreElements and nextElement methods of Query can be substituted by the methods hasNextSolutions and nextSolution, respectively. Finally, we print for every binding of X the names of the children of dietmar.

```
Variable x = new Variable("X");
Query q = new Query("father", new Term[] { new Atom("dietmar"), x });
while (q.hasMoreSolutions()) {
    Hashtable solution = q.nextSolution();
    System.out.println(solution.get("X"));
}
```

Listing 3.12: Calling PROLOG from JAVA with JPL.

The JAVA API of JPL has slightly changed with version 7.0.1 [92]. The Query class now implements the interfaces Iterable and Iterator. As a result, the methods hasMoreSolutions and nextSolution have been replaced by hasNext and next. In addition, the variable bindings from PROLOG are no longer saved as Hashtable but as java.util.HashMap. However, the overall approach of JPL has not changed with this recent version.
JPL uses the **Invocation API** of the JNI to dynamically instantiate or call public classes and methods which can be found at runtime. For this purpose, four predicates are provided by JPL:

- **jpl_new(+X, +Args, -V)**, with X to specify a class, array or primitive type, Args for possible constructor parameters as list and V bound to an object reference.

- **jpl_call(+X, +Method, +Args, -V)**, with X to specify a classname or type, the Method's name, Args for possible methods parameter as list and V for a possibly unified value.

- **jpl_set(+X, +Field, +V)**, with X to specify a a classname or type, the Field's name and an assignable, ground value V.

- **jpl_get(X, +Field, -V)**, with X to specify a classname or type, the Field's name and V for a possibly unified value.

JPL uses a simplified type system which allows for reference to any JAVA type based on a descriptor. The mapping from PROLOG to primitive or reference types in JAVA is fixed. Listing 3.13 shows a small JPL program in SWI-Prolog which instantiates a `javax.swing.JFrame` object with four numbered buttons.

```prolog
jframe_with_buttons :-
  jpl_new('javax.swing.JFrame', ['Frame with Buttons'], F),
  jpl_call(F, setLocation, [450,350], _),
  jpl_call(F, setSize, [400,200], _),
  jpl_new('java.awt.GridLayout', [2,2], G),
  jpl_new('javax.swing.JPanel', [G], Panel),
  maplist(button(Panel), ['1','2','3','4']),
  jpl_call(F, setContentPane, [Panel], _),
  jpl_call(F, setVisible, [@(true)], _),
  jpl_call(F, toFront, [], _).

button(Panel, X) :-
  concat_atom(['JButton ',X], Z),
  jpl_new('javax.swing.JButton', [Z], B),
  jpl_new('java.awt.Dimension', [40,40], D),
  jpl_call(B, setPreferredSize, [D], _),
  jpl_call(Panel, add, [B], _).
```

Listing 3.13: Calling JAVA from PROLOG with JPL.

A call of the `jframe_with_buttons` predicate in SWI-PROLOG displays the JAVA frame object as shown by Figure 3.1.
Discussion. JPl is only distributed together with SWI-PROLOG. It is not purely implemented in JAVA and makes extensive use of SWI’s Foreign Language Interface. Next to SWI-PROLOG, JPl has only basic support for YAPROLOG, an alternative PROLOG system.

The approach of JPl is an object-oriented representation of PROLOG’s language artifacts, and vice versa. Although there is an automatic type conversion of primitive types, the programmer has to laboriously address the more complex constructs of the foreign language. This leads to much boilerplate code in JAVA. With JPl, only a single PROLOG engine can be interfaced from JAVA. Ongoing issue with JPl are process termination from both sides as well as interrupts of pending calls. It is worth noting that not all methods of JPl are thread save which may lead problems in multi-threaded applications. In addition, JPl supports no modules in PROLOG. The transformation facilities for instances of user-defined JAVA classes to terms in PROLOG have to be implemented by the user. Moreover, the bindings of a solution of a query to PROLOG have the Term type and thus have to be first transformed to desired JAVA types. Most of the specifications that are necessary for the communication with PROLOG are based strings in JAVA. Therefore, IDE support via autocompletion or syntax highlighting is not available which clearly affects productivity.

3.3.3 PBR4J

In a former work [68], we dealt with the integration of business rules into JAVA. Business rules are statements that define or restrict certain aspects of a business process [14]. While business rules can occur purely informally, a careful, unambiguous, and consistent formulation of the rules is particularly important. Business rules can help to avoid cost-intensive misconceptions, to improve the communication, to ensure legal regulations, and to improve customer satisfaction [97]. Moreover, business rules can be used to separate business logic clearly from the rest of a business application. The business rules in [68] became part of a commercial enterprise resource planning
system for online merchants. One purpose of these business rules was to infer financial key data and costs for a given e-commerce scenario which depend on the market articles, the used online shopping platform and several parameters for the shipping. The derived business data supported the business intelligence module of the main application which was written in JAVA. For the integration of the business rules, we presented in [68] the integration framework Pbr4J (PROLOG Business Rules for JAVA).

**Description.** Pbr4J automates the integration efforts to some extend. It generates a JAR which contains the JAVA classes and methods to query a given set of business rules in PROLOG. In order to work with Pbr4J the business rules have to satisfy a single assumption. Every set of rules requires a single predicate which as entry point for the queries from JAVA. This predicate has to satisfy the structure name(Input, Output), where name denotes the name of the predicate and is usually associated with the purpose of the specific set of rules. The Input argument is a list of input facts and the Output argument is a list of output facts that are derived with the help of business rules in PROLOG.

In Pbr4J we distinguish between the knowledge base in the form of facts derived from objects in JAVA and the given set of business rules. The framework ensures that only conform and sound fact bases can transmitted from JAVA to PROLOG. Pbr4J uses XML Schema to specify the data exchange format. From this XML Schema, Pbr4J generates JAVA classes and methods which can be used to create and initialize a call to the business rules in PROLOG and to process the return from PROLOG. On the JAVA side, the necessary PROLOG terms are represented in an object-oriented manner. Therefore, the correctness and conformity of a sent fact base can be already verified on the JAVA side. The methods for the verification have been derived from standard methods for XML Schema. In addition, constraints on the composition of the fact base to that go beyond XML can also easily represented by rules in PROLOG, such as standard integrity constraints known from relational databases; e. g., primary key, foreign key, and not-null constraints.

However, the main contribution of [68] was not a new interface between PROLOG and JAVA but the simplified access to business rules in PROLOG. An important design decision has been not to mix PROLOG code with source code in JAVA. From the JAVA point of view, a query completely hides the fact that a set of rules in PROLOG is the target. Compared to JPL [89], it is not necessary to recreate PROLOG term structures with the help of particular PROLOG types in JAVA. With Pbr4J, PROLOG is considered as external domain-specific language (DSL) [31] for expressing knowledge in a declarative manner. Furthermore, Pbr4J proposes a clean separation between JAVA and the business logic in PROLOG. The PROLOG business rules can
be developed independently from JAVA and after the integration into JAVA, PBR4J ensures valid calls to PROLOG. In this way, PROLOG business rules can be integrated into and accessed from JAVA applications with minimal adaption effort in JAVA. In JAVA, the query and result handling are encapsulated by the generated classes which preserve the object-oriented programming style of JAVA.

Constraints between facts can be made explicitly by constraints which are formulated by particular rules in PROLOG. Listing 3.14 shows a foreign key constraint between the two facts shipping_charges/3 and tax/2. If a shipping_charges/3 fact is submitted without a tax/2 fact with matching Country argument, an exception is raised.

```
constraint(fk(shipping_charges)) :-
  forall(shipping_charges(Country, _, _),
      tax(Country, _) ).
```

Listing 3.14: Foreign Key Constraints.

The XML schema description for the fact base and the result set is extracted from templates for the input and the output of the called business rule in PROLOG. The extracted XML schema has to be extended manually by adding necessary mapping information for JAVA, such as the names of JAVA identifiers and types.

```
<x:s:element name="tax" type="tax_Type"
  minOccurs="1" maxOccurs="unbounded" />

<x:s:complexType name="tax_Type">
  <x:s:sequence>
    <x:s:element name="country" type="xs:string" />
    <x:s:element name="taxRate" type="xs:decimal" />
  </x:s:sequence>
</x:s:complexType>
```

Listing 3.15: Fragment of the XML Schema describing tax/2.

Listing 3.15 shows an XML schema fragment of the tax/2 fact. Every complexType element in the XML schema will be represented as separate class, the simpleType elements correspond to class members. Finally, PBR4J uses the XML schema to generate and compress the JAVA classes as JAVA archive (JAR). The JAR prevents the manual modification of the generated classes and simplifies the integration in JAVA.

Every JAR only contains the necessary classes to query an individual predicate that forms an entry point of a given set of business rules. A call from JAVA to the set of business rules of [68] is shown in Listing 3.16.
3.3 Integration via Communication Interfaces

```java
public class TestCall {
    public static void main(String[] args) {
        RuleSet rules = new RuleSet();
        KnowledgeBase kb = new KnowledgeBase();
        // ... fill the knowledge base with facts ...
        rules.query(kb);
        Iterator<FinancialKeyData> it = rules.getResultSet()
            .getFinancialKeyData().iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

Listing 3.16: A Java call to the business rules in Prolog.

Discussion. The Pbr4J approach has simplified the integration of PROLOG business rules into Java by the extensive usage of source code generation. A set of business rules in PROLOG is accessed via a single method call. The only parameter for a call to PROLOG is the passed knowledge base. Pbr4J uses a very basic mapping mechanisms for objects in Java to facts in PROLOG. In combination with the source code generation, the Java programmer is freed from PROLOG related 'glue' code and a manual transformation objects to terms. Therefore, the business rules in PROLOG can be accessed with a minimum of PROLOG-related knowledge. Moreover, the business rules in PROLOG can be developed independently from Java and then subsequently integrated into Java.

Nevertheless, the Pbr4J approach has several major issues. First of all, the Pbr4J approach is not very flexible. Every generated JAR only supports one kind of query to a given set of PROLOG business rules and the only parameter of such a query is the passed knowledge base. Moreover, Pbr4J relies on an intermediate data exchange format in XML which requires manual adaptions before the JAR can be generated with Pbr4J. Finally, backtracking in PROLOG is not controllable from Java with Pbr4J. Therefore, results from PROLOG cannot be obtained one by one.

3.3.4 LogicObjects

Combining new Java features with concepts of linguistic symbiosis, see also Section 3.2.2, seems to be a promising approach for an improved integration of logic and
object-oriented programming. LOGICOBJECTS [17] is a recent approach that combines concepts of linguistic symbiosis with modern JAVA features in order to overcome the limitations of former approaches. LOGICOBJECTS provides a transparent, (semi-) automated and customisable mechanisms for the integration of PROLOG and JAVA.

Description. In JAVA, the LOGICOBJECTS approach introduces symbiotic classes which are abstract JAVA classes. However, the implementation of symbiotic classes are transparently managed on the logic side. The issues of the object-to-term conversion of former approaches have been mitigated by a conversion of JAVA objects to objects in LOGTALK [53]. LOGTALK is an object-oriented extension to PROLOG and offers encapsulation features based on object-oriented concepts such as classes, objects and even prototypes. LOGTALK is implemented as preprocessor for common PROLOG compilers which compiles LOGTALK source code to plain PROLOG source code. A configuration file in plain PROLOG defines the interface between LOGTALK and a specific PROLOG compiler. An important design goal of LOGTALK has been compatibility with existing Prolog compilers and support for the ISO PROLOG standard. However, features outside the scope of the current ISO PROLOG standard such as dynamic predicates, syntax errors, and predefined operators are not supported by LOGTALK. In addition, there are also some known issues with operating systems.

With LOGICOBJECTS, the translation of instances of a symbiotic class to LOGTALK objects begins with the mapping of class name in JAVA to the corresponding object name in LOGTALK. Because LOGTALK objects transparently represent PROLOG predicates, we make no difference between both terms in the following description. The default mapping of JAVA class names includes the translation from JAVA’s CamelCase to snake_case in PROLOG. The default mapping is customizable in JAVA with the help of an annotation layer in JAVA. The @LObject annotation provides the name element to define an alternate target on the logic side. If the target is a parametric LOGTALK object, its parameters have to be specified by the args element of the @LObject annotation. In LOGTALK, a parametric object can be considered as a generic object. Instances of parametric objects are derived from given parameters.

For the translation of terms to objects in JAVA, the LOGICOBJECTS framework first tests for a symbiotic class with name and number of class members corresponding to the translated term’s functor and arguments, respectively. If different symbiotic classes match the term’s functor and arity, the framework simply returns the first match. However, this translation process can also be guided with the help of more annotations in JAVA, for instance the preferredClass element of the @LSolution
annotation overwrites the just mentioned rule of the first match. The described process is recursively applied to each arguments of a translated term.

In LOGICOBJECTS, the abstract methods of a symbiotic class are called symbiotic methods. The default mapping of symbiotic methods to predicates follows the same rules as the default mapping of symbiotic classes to LOGTALK objects. Symbiotic methods are mapped to predicates with functor and arity equals the method name and the number of the method parameters, respectively. This default mapping can be customized by the @LMethod annotation that has the same elements like @LObject.

If a symbiotic method has less parameters than the associated predicate arguments, the args annotation element of @LMethod can be used to indicate the position of the method parameter as argument. Although unbound variables are common in logic programming, java variables always have a value. In LOGICOBJECTS, unbound variables are defined as parameters of the args annotation element in plain PROLOG.

Listing 3.17 from [17] shows a symbiotic class definition that represents a subway line with a given name.

```
@LObject (args = \"name\")
public abstract class Line {
    String name ;

    public Line (String name) {this.name = name;}

    // answers if two stations are connected by this line
    public abstract boolean connects (Station s1 , Station s2);

    // answers the number of stations connected by this line
    @LMethod(name = \"connects\", args = \"_\", \"_\")
    public abstract int segments();
}
```

Listing 3.17: The symbiotic class Line in LOGICOBJECTS.

By default, the symbiotic line class maps to the parametric line object in LOGTALK with name as object parameter which is shown in Listing 3.18 from [17].

```
:- object(line(_Name)).
:- public([connects/2]).

connects(S1, S2) :- self(Self), metro::connected(S1, S2, Self).
:- end_object.
```

Listing 3.18: The parametric line object in LOGTALK.
The annotation elements name and args indicate that any invocation of the symbiotic method segments is delegated to the connects predicate with two anonymous variables ("_") as arguments.

To derive the return values of symbiotic methods LOGICOBJECTS relies on different heuristics. By default, the type corresponding to the first solution found on the logic side is considered to be the return value of the symbiotic method. The first heuristic for result types is based on a naming convention. If an unbound variable in a query is named LSolution, the binding of LSolution is considered to be the representation of the return value as term. This term is then translated to a Java object as already discussed above. If no LSolution variable is not present, the LOGICOBJECTS framework tries to infer the return type by the signature of the invoked method. Then the representing of the return value as term has the name of the invoked method as functor and the method parameters as arguments. Finally, the programmer can use the @LSolution annotation to explicitly specify the representation of the return value as term.

In Java, the return type of a method already defines if a single value is expected or an entire collection of values. In Prolog, however, a query may have one solution which is a list of terms or multiple solutions due to backtracking. To distinguish both cases LOGICOBJECTS provides the @LComposition annotation for symbiotic methods which specifies the return type as collection of multiple solutions.

The LOGICOBJECTS frameworks provides the factory class LogObjectFactory with a create method to instantiate a symbiotic class. The parameters of the create method are a class object of the symbiotic class which we want to instantiate followed by all of its constructor parameters. The instantiation mechanism of LOGICOBJECTS requires also source code generation and byte code instrumentation at runtime.

Discussion. Symbiotic classes and methods provide a tight coupling of PROLOG and JAVA. However, additional knowledge and implementation effort is required for the intermediate layer with LOGTALK and the coupling of symbiotic classes and objects in LOGTALK has to be maintained manually. Furthermore, the known issues of LOGTALK equally apply to LOGICOBJECTS. Even though LOGTALK has been designed for compatibility with several PROLOG compilers, not all PROLOG compilers are supported. Moreover, the integration of PROLOG predicates in JAVA requires duplicate effort. Corresponding to the definition of the symbiotic classes in JAVA, predicates in PROLOG have to wrapped by LOGTALK objects. The instantiation of symbiotic classes requires source code generation and byte code instrumentation at runtime which affects the performance. Moreover, the usage of the factory method is error-prone as a JAVA programmer has to remember and arrange correctly all
necessary constructor parameters for the instantiation of a symbiotic class. Auto-
completion is not available for this task and errors only occur at runtime.

Because of the relational character of predicates in PROLOG, the mapping of JAVA
methods to PROLOG predicates leads to multiple but very similar methods in JAVA
which all have the same predicate in PROLOG as target. In order to integrate a PRO-
LOG predicate with three arguments that may act either as input or output, eight
symbiotic methods in JAVA would be required to cover each use case. Moreover,
because a JAVA method has only a single return type, symbiotic methods cannot be
used to integrate predicates with two or more arguments for the output. In addition,
symbiotic methods provide no explicit control over backtracking in PROLOG, they
either return one solution or a collection of all solutions.

The performance tests with LOGICOBJECTS in [17] reveal also performance issues.
Compared to the custom JAVA-PROLOG interface JPL of SWI-PROLOG, the tests
with LOGICOBJECTS have been in the worst case 23 times slower. This is consid-
erable, in particular, because the tests with LOGICOBJECTS have used JPL for the
communication between JAVA and SWI-PROLOG.

3.4 Alternative Approaches

The presentation of approaches for the integration of logic and object-oriented pro-
gramming has not been exhaustive, so far. In the following, we outline some alter-
native approaches.

An alternative translator not for PROLOG but for object-oriented logic program-
ming language ACTOR PROLOG [54] is described in [55]. Similar to LOGTALK, the
ACTOR PROLOG programming language combines aspects of objected-oriented pro-
gramming such as classes, objects (which are called worlds in ACTOR PROLOG), and
inheritance with logic programming. The declarative semantics of ACTOR PROLOG,
without non-logical built-in predicates, are the semantics of PROLOG. In contrast
to LOGTALK, classes and worlds are only syntactic sugar and have purely declar-
ative semantics. Moreover, ACTOR PROLOG provides an extended control strategy
based on the repetitive proof of logical actors which are particular subgoals that
share common variables. However, the translator of [55] is only applicable to pro-
grams in ACTOR PROLOG and suffers from similar issues as already discussed for the
translators jPROLOG and PROLOG CAFE. The interpretation of the translated pro-
gram in JAVA is very difficult and the intermediate source code feels not natural for
JAVA programmers. The translation adds a considerable layer to the object-oriented
program. Because the JVM provides no run-time optimization methods for logic pro-
grams, only a static source code optimization is possible. Moreover, the benchmarks
in [55] show that the translated Actor Prolog programs perform up to 8 times slower than comparable programs in SWI-Prolog.

An alternative Prolog system embedded in Java is the JLog [41] interpreter. The JLog interpreter is a fairly complete implementation of Prolog. However, there are some discrepancies, for instance, JLog does not support exceptions or file, stream, and socket based I/O, just to name a few. Queries from Java to JLog usually are expressed in string-format. Similar to P@J for the embedded Prolog engine TuProlog from Section 3.2.1, also JLog includes basic translation facilities to map between primitive Java types, Java strings, and standard Java objects and JLog terms. The benchmarks in [28] show that JLog is significantly slower than SWI-Prolog. For instance, the 22-Queens problem\(^9\) has been solved at least 22 times slower than in SWI-Prolog.

An alternative communication interface between SWI-Prolog and Java is based on network sockets. The Prolog Development Tool (PDT) [78] provides the Prolog Connector library [79] in Java which allows to query multiple SWI-Prolog processes. This client-server relation is based on sockets. The PDT is a Prolog IDE which is distributed as plug-in for the Eclipse platform. The Prolog Connector library has only two Java methods to execute a query to Prolog. The \texttt{queryOnce} method only returns the first solutions of a query and the \texttt{queryAll} method returns a collection of all solutions. All variable bindings are returned as instances of the most general Java type \texttt{Object} which leads to subsequent type casts. Similar to Interprolog, queries from Java to Prolog are defined purely in string format. Alternatively and to avoid string concatenations, the \texttt{buildTerm} method offers a simple term construction mechanism which allows us to pass a term’s functor and arguments in string format as method parameters. All the issues that we already have discussed for Interprolog and JPl in the Sections 3.3.1 and 3.3.2, respectively, are equally true for the Prolog Connector library. Moreover, the Prolog Connector library provides no control over backtracking in Prolog.

3.5 Synthesis

In this section, we constitute the motivation for the thesis at hand. We already have presented numerous approaches for the integration of Prolog and Java and the attentive reader may wonder what is actually missing?

\(^9\)The \(n\)-Queens problem is described as the placing of \(n\) non-attacking queens on a chessboard with \(n \times n\) squares. It can be shown that for all natural numbers \(n\), except for \(n = 2\) and \(n = 3\), there exist a solutions.
3.5.1 Discussion

First of all, we can observe that almost all the presented approaches work only together with a specific PROLOG system which complicates an exchange of the used PROLOG system, be it an embedded or non-embedded in JAVA. An exchange usually requires major adaption in JAVA. Moreover, we want to exploit from JAVA different PROLOG systems as they offer different features and program libraries for various domains. However, the modalities for querying PROLOG from JAVA should always remain the same. Therefore, we wish for an uniform approach that is freely available.

The presented approaches for the integration of PROLOG and JAVA have been broadly categorized into the following three types:

1. Translation of PROLOG programs to JAVA.
2. Embedding of PROLOG in JAVA.
3. Communication interfaces between a JAVA runtime and a PROLOG process.

The translation of PROLOG programs to JAVA as we have already discussed has significant disadvantages. The translated programs usually feel not natural to JAVA programmers and add a significant layer to the object-oriented program which also cannot be optimized by the JVM. Moreover, the translation of larger PROLOG programs to JAVA is very complex and cannot be executed in a straightforward manner. For instance, built-in predicates of specific PROLOG systems used in the PROLOG program cannot be translated to JAVA as their implementation is not written in PROLOG. The same is true if extensions to PROLOG are involved or if the used program libraries incorporate native code. Another aspect is that PROLOG programs that have been translated to JAVA perform usually worse in JAVA than the original PROLOG program executed by a PROLOG system which is written in C or C++. All these issues suggest that the compilation of PROLOG programs to JAVA is not a satisfactory option for an uniform approach. The last two types rely on a communication based on messages between either objects in JAVA or a PROLOG process and a JAVA runtime. Therefore, we wish for a portable, uniform approach that does not limit the power of the various PROLOG systems. Such an approach cannot rely on a translation of entire PROLOG programs to JAVA. Instead, it has to support the message-based communication between PROLOG and JAVA.

Due to the popularity and omnipresence of JAVA in the last two decades the focus shifted to an improved object-orientated perspective of the integration of PROLOG and JAVA. Programming in PROLOG has been partially incorporated explicitly or implicitly in the JAVA programming language. However, the blend of programming
concepts and semantics of PROLOG with JAVA has its price. For instance, logic extensions to JAVA that are based on a modified JVM result in less portable programs which violate the JAVA idiom 'write once, run everywhere'. Another aspect is a reduced accessibility. Do we really want to learn a third programming language in order to combine PROLOG and JAVA? Why would we abandon the clear and concise programming style in plain PROLOG which has been the actual reason for an integration? Former approaches [89, 21, 17] have introduced abstraction layers in JAVA which support the object-oriented perspective of the integration of PROLOG and JAVA. However, these abstraction layers are often less accessible and readable.

Queries to PROLOG lead to lots of boilerplate code that obfuscates the actual intention. In addition, even for simple queries a deeper knowledge for PROLOG data structures is mandatory. This poses a noteworthy obstacle for many JAVA programmers which are often little experienced with PROLOG. On the other hand, queries to PROLOG in plain string format [15, 79] are no viable solutions as they are not object-oriented and error-prone. Moreover, strings in JAVA are not supported by autocompletion or syntax highlighting which affects the productivity. Therefore, we wish for an uniform approach that supports a clean separation of PROLOG and JAVA programs and keeps the power of the individual programming languages untouched.

In JAVA, we wish for an object-oriented query mechanism to PROLOG that is clear and concise. The query format has to feel natural for JAVA programmers.

In addition, we require sophisticated mechanisms that significantly reduce the programming effort for the integration of PROLOG functionality with JAVA and provide an object-oriented interface to associated predicates in PROLOG. To simplify the interaction with PROLOG we wish also for an automated conversion mechanism for objects and terms. Instead of a static approach, we wish for customizable conversions which are not limited to plain JAVA objects [41, 21] but support the conversion of complex, user-defined JAVA types.

However, any novel approach has to solve the following essential interoperability problems on the object-oriented perspective of the integration of PROLOG and JAVA.

1. Unbound variables such as in PROLOG have no equivalent in JAVA. How are they specified for queries to PROLOG?

2. Methods in JAVA have fixed parameters for the input and a single fixed return type. In PROLOG, arguments of predicates may either act as input parameter or as results. In this way, predicates in PROLOG are also not limited to a single return value. How is the flexibility of predicates in PROLOG resolved in JAVA?
3. Depending on backtracking, queries to PROLOG may lead to no solution, one solution, or to multiple solutions. How is the non-determinism of queries to PROLOG resolved in JAVA?

Finally, we wish that all applied techniques, be it for abstraction or automatization, have no significant affect on the runtime performance in JAVA.

3.5.2 Important Criteria for a Seamless Integration

As a result of our discussions throughout this chapter, we have constituted several important criteria for a seamless integration of PROLOG and JAVA from the object-oriented perspective. Any novel approach for the integration of PROLOG and JAVA should satisfy all of them and provide suitable solutions to the previously mentioned interoperability problems. The criteria listed in the following have been broadly categorized by important aspects of software engineering.

[A] Accessibility

[A1] Concise and object-oriented queries from JAVA to PROLOG.


[A3] Non-commercial, open source license.

[C] Compatibility

[C1] No dependency on a particular PROLOG system.

[C2] No modifications or extensions to the JAVA Virtual Machine.

[C3] No translation of PROLOG programs to JAVA.

[C4] No dependency on native code.

[E] (Programming) Effort

[E1] Simplified integration of PROLOG predicates into JAVA.

3 Integration Approaches for Prolog and Java

[F] Flexibility
[F2] Support for non-determinism.
[F3] Context-dependent return types for queries to PROLOG in JAVA.

[O] Obfuscation
[O1] No boilerplate code for queries to PROLOG in JAVA.
[O2] Object-oriented interface for predicates in PROLOG.
[O3] Clean separation between programming in PROLOG and JAVA.

[P] Productivity
[P1] Decent execution times for the routines in JAVA.
[P2] Autocompletion and syntax highlighting for queries to PROLOG in JAVA.
[P3] At compile time verifiable source code for queries to PROLOG in JAVA.
[P4] Error localization in PROLOG and JAVA.

This list makes no claims for completeness. However, we have tried to cover all essential aspects which contribute significantly to a seamless integration of PROLOG and JAVA from the object-oriented perspective. In the following Chapter 4, we propose our object-oriented, uniform approach for the integration of PROLOG and JAVA. A comparison of the criteria listed with our own approach and the discussed related work follows in Section 8.3.2.
4 The Connector Architecture CAPJa

Modern software is often realized as a modular combination of subsystems for, e.g., knowledge management, visualization, verification, or the interaction with users. As a result, software libraries from possibly different programming languages have to work together. Even more complex the case is if different programming paradigms have to be combined. This type of diversification of programming languages and paradigms in just one software application can only be mastered by mechanisms for a seamless integration of the involved programming languages. However, the integration of the common logic programming language PROLOG and the popular object-oriented programming language JAVA is complicated by various interoperability problems which stem on the one hand from the paradigmatic gap between the programming languages, and on the other hand, from the diversity of the available PROLOG systems. The subject of the thesis is the investigation of novel mechanisms for the integration of logic programming in PROLOG and object-oriented programming in JAVA. We are particularly interested in an object-oriented, uniform approach which is not specific to just one PROLOG system.

The main contribution of the thesis is a novel integration framework called the Connector Architecture for Prolog and Java (CAPJa) which provides an object-oriented, uniform approach to the integration of PROLOG and JAVA. The framework is completely implemented in JAVA and imposes no modifications to the JAVA Virtual Machine or PROLOG. CAPJa provides a semi-automated mechanism for the integration of PROLOG predicates into JAVA. For compact, readable, and object-oriented queries to PROLOG, CAPJa exploits lambda expressions with conditional and relational operators in JAVA. The communication between JAVA and PROLOG is based on a fully automated mapping of JAVA objects to PROLOG terms, and vice versa. In JAVA, an extensible system of gateways provides connectivity with various PROLOG systems and, moreover, makes any connected PROLOG system easily interchangeable, without major adaption in JAVA.

The purpose of this chapter is to give the reader an overview on CAPJa. We begin in Section 4.1 with a description of core features of CAPJa and associated components. In Section 4.2, we focus on typical applications with CAPJa and characterize the common phases of the development with CAPJa.
4.1 Components

Core features of CAPJa can be summarized as follows:

- An automated mapping mechanism for objects in JAVA and terms in PROLOG.
- A semi-automated integration mechanism for PROLOG predicates.
- An elegant and powerful query mechanism for PROLOG.
- Connectivity of JAVA with various different PROLOG systems.

CAPJa has three main components, see also Figure 4.1, which provide all the core features from above as follows:

- JPMAPPING defines and regulates the mapping of objects in JAVA to terms in PROLOG, and vice versa. Furthermore, it provides the mechanism for the semi-automated integration of PROLOG predicates into JAVA.
4.1 Components

- **JPLambda** offers in Java a powerful query mechanism to Prolog and allows for the definition of custom object-to-term mappings for Java types for which the source code is not accessible.

- **JPGateway** provides a systems of gateways to Prolog which connects Java with different Prolog systems. Custom gateways can easily be added in order to support various communication interfaces.

In the remainder of this section, we introduce each of CAPJa’s components shortly. A detailed description of each component follows in the subsequent Chapters 5, 6, and 7, respectively.

**JPMapping.** CAPJa relies essentially on an automated object-to-term mapping (OTM) which derives from a Java object a representation as Prolog term. The JPMapping component regulates the whole process internally. For instance, if we want to assert a Java object as fact to Prolog, JPMapping derives a representation as term in Prolog which is then asserted to Prolog’s internal database. In Java, this is only a single method invocation with plain Java object as parameter.

The mapping mechanism is highly flexible. The representation of a Java object in Prolog can be customized conveniently with the help of a special source code annotation layer (@JPMapping) which encodes the relevant specifications. In addition, the JPLambda component provides an alternative for custom mapping definitions based on particular lambda expressions in Java and is also applicable Java types for which the source code is not accessible, for instance only the class files are provided as Java Archive (Jar). The annotations as well as the lambda expressions allow to define multiple mappings for an individual Java type. If no mapping is not stated explicitly, CAPJa falls back automatically to an internal default mapping mechanism. The default mapping mechanism translates a given Java object to a Prolog term that has the object’s the class name as functor and the class members lexically ordered as arguments. A custom mapping can also be marked as default mapping for given a Java type which notifies CAPJa to use the marked custom mapping as default mapping instead.

The Predicate-Signature Notation (PSN) is a small domain-specific language (DSL) in Prolog that is used to characterize the signatures of predicates in Prolog. From predicate signatures in PSN, CAPJa automatically generates an object-oriented interface in Java that consists of classes that associate the predicates which have been characterized with the PSN in Prolog. Instances of the generated classes now map by default to the characterized predicates in Prolog. In this way, we can easily address the Prolog predicates from Java and the object-to-term mapping
is applied automatically. Figure 4.2 summarizes the features of the JPMAPPING component.

The representation of a Java object as term is created during runtime by subclasses of the generic abstract class JPMapper<T>. The type parameter T denotes the Java type for which the subclass applies. Throughout this work, we call T simply the mapping type and a subclasses of the abstract class JPMapper<T> just mapper. Every mapper provides methods for the translation of instances of its mapping type to Prolog, and vice versa. The source code of mappers never has to be written manually. A mapper is generated during an extended build process in Java which has to be applied once, just before a program developed with CAPJa is executed for the first time. During the build process, the custom source-to-source translator JPCompiler derives the implementation of a mapper automatically from either a @JPMapping annotation or a lambda expression for custom mappings and generates the corresponding Java source code. If CAPJa requires to map a Java type for which no mapping has been defined, JPCompiler derives and generates automatically a mapper that implements CAPJa’s default mapping for the given Java type.

JPLambda. The JPLAMBDA component supports in Java the creation of queries to Prolog. The JPQuery<T> class encapsulates any query to Prolog in JPLAMBDA. Throughout this work, we call the type parameter T of JPQuery<T> just the query type. A predicate in Prolog can simply be called from Java with the help of a JPQuery<T> instance with a query type that either is a generated class associated to the predicate via the PSN or any Java type for which a mapper exists that has the given predicate as image of the mapping. The methods of JPQuery<T> can be used to query for the existence of a solution, for one solution, or for all solutions. In addition, they provide control over backtracking in Prolog. Solutions can be obtained one by one from Prolog. A solution returned from Prolog is a new instance of the query type.
Queries to \textsc{prolog} can be extended by particular query constraints which are added as subgoals to a query to \textsc{prolog}. Constraints are formulated with the help of a DSL embedded in \textsc{java} which we call the \textit{java-prolog query language} (\textsc{jpql}). All query constraints of a single query are represented in \textsc{java} by a single Boolean lambda expression which is passed as constructor parameter of \textsc{jpquery}<\textsc{t}>. The lambda consists of Boolean expressions that include expressions based on relational operators and that are linked by conditional operators. The Boolean expressions has to be associated with a member of the query type. Query constraints are translated to subgoals of the initial query or to bindings of logical variables in the actual query. For instance, equality equations are translated to a variable binding and inequalities are translated to subgoals that associate arguments of the addressed predicate in \textsc{prolog}. More query constraints supported by \textsc{jpql} are described later in Section 6.2. The \textsc{jpql} only serves as a specification language for queries to \textsc{prolog}, the lambda expressions used for query constraints have no runtime behavior. Similar to the \texttt{@\textsc{jpmapping}} annotations for mappers, the \textsc{jpql} is analyzed by \textsc{jpcompiler} during the extended build process. The constraints of a query are split into two sets: query constraints that can be translated to \textsc{prolog} and query constraints that cannot be translated to \textsc{prolog}. Translated query constraints are submitted to \textsc{prolog} along with the actual query. Query constraints that cannot be translated to \textsc{prolog} are subsequently tested in \textsc{java}. \textsc{jpcompiler} analyzes source files with instances of \textsc{jpquery}<\textsc{t}> and derives an abstract syntax tree (\texttt{ast}) for each query. From the \texttt{ast}, \textsc{jpcompiler} generates a subclass of \textsc{jpquerytranslator}<\textsc{t}> which is linked in the source file with the analyzed query. The subclass of \textsc{jpquerytranslator}<\textsc{t}> efficiently implements the actual query together with the query constraint and is internally used to query \textsc{prolog}. Finally, all the generated and modified source files are compiled again.

In addition to the \textsc{jpql}, the \textsc{jplambda} component provides the \textit{java-prolog mapping language} (\textsc{jpml}) which is also a small DSL embedded in \textsc{java}. Similar to the \textsc{jpql}, the \textsc{jpml} uses lambda expressions to define custom object-to-term mappings. In so doing the \textsc{jpml} associates a predicate signature with a given \textsc{java} type. In contrast to custom object-to-term mappings with \texttt{@\textsc{jpmapping}} annotations, the \textsc{jpml} can be applied to \textsc{java} types for which the source code is not accessible; e. g., only the class files of a \textsc{jar} are available. The \textsc{jpml} is similarly processed by \textsc{jpcompiler} as the \textsc{jpql}.

\textbf{jpwgateway.} An important design goal for \textsc{capja} is compatibility. Therefore, the \textsc{jpgateway} component provides connectivity with different \textsc{prolog} implementations and separates \textsc{java} cleanly from \textsc{prolog}. The \textsc{jpgateway} component uses a system of gateways which handle the specifics of a connected \textsc{prolog} systems.
A gateway implements a specific communication interface to PROLOG. Every gateway is a subtype of the Java interface JGateway which forms the generic link to the other components of CAPJa. In all classes of JPMAPPING and JPLAMBDA, only the JGateway interface is referenced.

The JGateway component offers a default gateway which is called the Portable Prolog Gateway (PPG). The PPG is not specific to any particular PROLOG implementation and uses the standard streams for input and output for the communication with a PROLOG process. In this way, the PPG connects various PROLOG systems with JAVA and even supports multiple PROLOG engines. Compatibility has been tested successfully with several freely available PROLOG systems such as BPROLOG, the CIAO system, GNU-PROLOG, SWI-PROLOG, TU-PROLOG, the XSB system, and YAPROLOG. More gateways can easily be added with little effort. Every new gateway has only to implement the abstract methods of the JGateway interface. In Section 7.4.1, we propose gateways for the communication with PROLOG via sockets, HTTP, and TCP. Already established JAVA-PROLOG interfaces such as Jpl for SWI-PROLOG can be exploited as gateways, too. In that case, CAPJa works on top of JPL as a high-level API in JAVA.

4.2 Development with CAPJa

CAPJa can be easily integrated into any JAVA project, for instance as JAR that contains the necessary binaries. Alternative ways for the integration of CAPJa are described later in Section 8.3.2 and 9.2. Depending on the gateway used some configurations have to be added in the form of a configuration file. For instance, CAPJa’s default gateway, the PPG, works with a configuration file in XML that contains, e.g., the location of the PROLOG executable in the local file system which is necessary to start a PROLOG process from JAVA.

Typically, there are three main scenarios for the development with CAPJa:

- Queries from JAVA to the PROLOG database.
- Computations in PROLOG based on data from JAVA.
- Modifications to the PROLOG database from JAVA.

The development with CAPJa usually can be divided into three subsequent phases:

1. The integration of PROLOG predicates into JAVA.
2. The definition of custom object-to-term mappings in JAVA.
3. The creation of queries from JAVA to PROLOG.
Predicate Integration. To integrate PROLOG predicates in JAVA, their signatures first have to be characterized in Psn. From the signatures in Psn, CAPJa generates an object-oriented interface for JAVA together with all the necessary mappers. For JAVA types with suitable properties for a mapping to a given predicate in PROLOG, we can define custom object-to-term mappings. It should be noted that it is not possible to combine the properties of different classes within a single mapping. CAPJa relies on a one-to-one relation between JAVA classes and predicates in PROLOG. However, the properties of different classes still can be used for an object-to-term mapping if they are combined by a single adapter class that instead associates the predicate in PROLOG.

Custom Object-To-Term Mappings. Once there is a fixed one-to-one relation between the predicates in PROLOG and classes in JAVA, we can deal with the mappings of objects in JAVA to terms in PROLOG. For classes that have been generated during the previous phase, no mappings have to be defined, CAPJa already has generated the corresponding mapper classes. Custom mappings are defined in JAVA only. At this point, we want to mention that also classes can be mapped to PROLOG which have no associated predicate in PROLOG. A valid use-case is to map and assert JAVA objects in the form of facts to PROLOG’s internal database. These facts then can be queried from JAVA as usual with the JPQL and form a fact base for further computations in PROLOG.

Let \( p \) be a predicate in PROLOG which corresponds to a class \( C \) in JAVA. If \( C \) and \( p \) comply with CAPJa’s default mapping, than no mapping needs to be configured. CAPJa automatically generates a mapper that implements the default mapping for \( C \). If \( C \) and \( p \) do not comply with the default mapping, then we require a custom mapping. For instance, if the class name and the predicate’s functor do not match, the number of class members of \( C \) differs from the number of arguments of \( p \), or the lexical order of the class members does not match the order of the arguments of \( p \). Then, we have two options in JAVA to configure a suitable mapping. If the source code of \( C \) is accessible, we can simply add a \( @JPMapping \) annotation to \( C \) which characterizes the mapping. Otherwise, we can define the mapping for \( C \) with the help of Jpml. Again, it is important to note that no mapper has to be implemented manually. CAPJa generates the necessary source code either from the specifications in Psn, \( @JPMapping \), or Jpml. All three options are clear, concise and only require little programming effort.

Construction of Queries. If for all relevant classes and predicates mappings have been defined, we can use the JPQL to query PROLOG from JAVA. As mentioned before, we express queries to PROLOG by an instance of the JPQuery\(<T>\) class. Almost
any Java type is a valid query type, especially those for which exist an one-to-one relation with a predicate in Prolog. If a query type is not associated with a predicate in Prolog, we still can query for instances of the query type. Usually, these instances have previously been asserted to Prolog in the form of facts.

Any query can be extended by particular query constraints which associate the properties of the query type. For instance, if we query for persons and want to limit the set of resulting persons, we can define a query constraint that is associated with a property of person, for instance the person’s age. The query type determines also the lambda parameter in Java which can be elegantly used as template, on top of which we define the query constraints. Multiple query constraints can be connected with conditional operators and, similar to Prolog, precedence between query constraints can be expressed by round brackets. Queries in JPQL are completely object-oriented and, due to the code as data property of lambda expression, they are clear and concise. An additional positive effect is that autocompletion and syntax highlighting remain completely operative in JPQL. Depending on the used method for the query to Prolog, the return type either corresponds to the query type, a collection of the query type, or an iterator [34] over a collection of the query type. No type casting or conversion is necessary for the returned variable bindings of Prolog. If we query Prolog for persons, CAPJa returns objects that represent persons in our Java application.

However, some calls to some special predicates such as assertz/1, retract/1, or consult/1 are executed by particular methods of the used gateway, no instance of JPQuery<T> is needed. The parameters of these particular methods are either plain Java objects or a path expression in order to load a Prolog file. The implementation of these methods entirely depends on the used gateway.
5 The JPMapping Component

For the integration of programming languages, data exchange is mandatory. Because language artifacts of PROLOG and JAVA differ, a mapping mechanism is required to overcome the paradigmatic gap. In Chapter 3, we already have discussed several approaches for mappings between PROLOG and JAVA. Identified issues of former approaches have been boilerplate code, implicit translation effort for the programmer, lack of control and flexibility, significant overhead at runtime, and dependencies on particular PROLOG systems. In this chapter, we present our own approach which is the JPMapping component of CAPJa.

The JPMapping component offers an automated, customizable mapping mechanism between JAVA objects and PROLOG terms. The mapping mechanism forms the basis for the advanced features of CAPJa which we will discuss in Chapter 6. In JAVA, JPMapping provides a mechanism that is based on a default mapping which applies to almost any JAVA class, without additional programming effort. Manual adaptions or the creation of additional source code in JAVA is not necessary. In addition, the default mapping can be modified by source code annotations in JAVA which allow us to completely customize the representation of class instances in JAVA as terms in PROLOG. The annotation layer is only used for the mapping specifications and it is not reflected to guide the mapping process at runtime. Instead, JPMapping provides a source code generator that creates for (annotated) JAVA types corresponding mapper classes which efficiently implement the necessary conversion facilities between objects and terms.

In PROLOG, we introduce the Predicate-Signature Notation (PSN) which is used to characterize the signatures of predicates in PROLOG. By predicate signature we mean a description of the structural composition of a predicate, for instance a PROLOG compound is characterized by its functor, arity, and arguments. Also the predicate signature in PSN can be passed to the source code generator of JPMapping in order to create mapper classes in JAVA. Next to the mapper class, an additional new JAVA type is generated which has been derived from predicate signature in PSN and represents the characterized PROLOG predicate in JAVA. The conversion facilities of the generated mapper only translate the instances of the new JAVA type to the characterized PROLOG predicate, and vice versa. In this way, JPMapping provides a powerful predicate integration mechanism.
The remainder of this chapter is organized as follows: Section 5.1 describes the object-to-term mapping mechanism of JPMapping. Section 5.2 introduces the PSN and Section 5.3 explains how actual object-to-term mappings are finally implemented and how their source code is generated.

5.1 Object-To-Term Mappings

In this section, we describe CAPJa’s mapping mechanisms between objects in Java and terms in Prolog. We begin with an introductory example of the object-to-term mapping in CAPJa in Section 5.1.1, which is followed by a formal definition of the object-to-term mapping in Section 5.1.2. In Section 5.1.3, we explain the default mapping mechanism of CAPJa which has been used in this introductory example for the translation of the Student instance bart into the Prolog term in Listing 5.3. In Section 5.1.4, we discuss how a default mapping can be modified by particular source code annotations in Java.

5.1.1 Introductory Example

Listing 5.1 shows the definitions of a Person class in Java which is extended by a Student class. We omit the straightforward definitions of constructors, getter, and setter.

```java
1 class Person {
2     public String firstName;
3     public String lastName;
4     // constructor, methods...
5 }
6 class Student extends Person {
7     private int age;
8     private ArrayList<Course> courses = new ArrayList<>();
9     private int id;
10     // constructor, methods...
11 }
12 class Course {
13     private int id;
14     private String name;
15     // constructor, methods...
16 }
```

Listing 5.1: Sample data structures in Java.
In the following Listing 5.2 we have created the Student instance bart and added a Course instance c. The course name of c is not set explicitly. Because the name field of the Course class is of type String, the corresponding default value is null in JAVA.

```java
Student bart = new Student("Bart", "Stimpson");
bart.setAge(10);
Course c = new Course(897);
bart.getCourses().addCourse(c);
bart.setId(181353);
```

Listing 5.2: Instance bart of Person in JAVA.

The default mapping mechanism of CAPJa then translates the bart object to the PROLOG term in Listing 5.3.

```prolog
student(10, [course(897, n_u_l_l)], "Bart", 181353, "Stimpson")
```

Listing 5.3: Representation of the bart object as term in PROLOG.

The fields age, courses and id of the Student class together with the inherited fields firstName and lastName become the arguments of a student/5 predicate which are sorted in lexical order according to their names in JAVA. The courses field of the Student class is a list-type with a non-primitive type parameter and thus CAPJa’s default mapping recursively applies to each member of the list. Because the name field of the course instance is null, it is mapped to an user-defined atom which is n_u_l_l in our example and uniformly represents null references in JAVA.

5.1.2 Formal Definition

Let $\mathcal{C}$ be the set of all given classes, $\mathcal{P}$ be the set of all primitive JAVA types and $\mathcal{I}_\mathcal{P}$ be all possible instances of primitive JAVA types. Then, a JAVA class $c$ can be defined as the tuple $(p_c, n_c, H^l_c \times F^m_c, M_c) \in \mathcal{C}$, with package $p_c$ and name $n_c$, fields $H^l_c \times F^m_c$, and the set $M_c$ of methods accessible from $c$. Inheritance in JAVA is an object-oriented concept that allows us to pass properties and methods from a superclass to a subclass. In this way, a class hierarchy is formed by a transitive ‘is-a’ relation between superclass and subclass. In JAVA, inheritance between classes is indicated by the extends keyword in the class declaration. A subclass in JAVA inherits visible (public, protected) fields and methods from its superclass. Therefore, for a class $c$ we distinguish the inherited fields $h_i, i = 1, ..., l$ from class fields $f_j, j = 1, ..., m$ and the set $H^l_c \times F^m_c$ can be expressed as tuple $(h^l_c, f^m_c)$ with $h^l_c \in H^l_c \subset \mathcal{C} \cup \mathcal{P}$ and $f^m_c \in F^m_c \subset \mathcal{C} \cup \mathcal{P}$. 

83
Let $I_c$ be the set of all instances of the class $c \in C$. In the following, two objects $o_1^c, o_2^c \in I_c$ are said to be equivalent, denoted by $o_1^c \equiv o_2^c$, if, and only if, they have exactly the same fields and field values. The equivalence class of equivalent objects is denoted by $\overline{o}$ and the set of all equivalence classes $o_c$ is denoted by $I_c$. The equivalence class $o_c$ can be represented as tuple $(p_c, n_c, H^c_{c}(o) \times F^m_{c}(o), M_c) \in I_c$, with $H^c_{c}(o) \times F^m_{c}(o) := (h^c_1(o), ..., h^c_i(o), f_1^c(o), ..., f^c_n(o))$ and $h^c_1(o) \in F^c_o \subset I_c \cup I_P$ and $f^c_j(o) \in F^c_o \subset I_c \cup I_P$. In the following, we simply refer to $o_c$ as object.

Let $T$ be the set of all given PROLOG terms. A PROLOG term $t^c_o$ corresponding to an object $o_c$ is defined as relation $t(a^c_1(o), ..., a^c_n(o)) \in T$, with functor $t$ and arguments $a^c_k(o) \in T, k = 1, ... n$.

The mapping of objects to terms then can be written as function $d : I_c \rightarrow T$ of the set of all class instances to the set of all PROLOG terms. The image of an object $o_c$ under $d$ satisfies the following equation:

$$d(o_c) = d(p_c, n_c, H^c_{c}(o) \times F^m_{c}(o), M_c) = d(p_c, n_c)d(H^c_{c}(o) \times F^m_{c}(o))$$

$$= d(p_c, n_c)(d(h^c_1(o)), ..., d(h^c_i(o)), d(f_1^c(o)), ..., d(f^c_n(o))) = t(a^c_1(o), ..., a^c_n(o)). \quad (5.1)$$

Note that the methods $M_c$ of an object $o_c$ are not mapped at all under $d$. This is intentional. From our point of view, there is no need to map methods in JAVA to predicates in PROLOG. The reason for this is that we do not map functionality from JAVA to PROLOG. As Equation 5.1 suggests the mapping $d$ can be split into the two mappings $d(p_c, n_c) = t$ and $d(H^c_{c}(o) \times F^m_{c}(o)) = (a^c_1(o), ..., a^c_n(o))$.

**Mapping** $d(p_c, n_c) = t$ of Class Name and Package. Package and class name are translated to simple atoms in PROLOG. Dots in a package name $p_c$ are replaced by single underscore characters; e.g., *my.new.package* is translated to *my_new_package*. This improves readability and since dots usually are not allowed in a PROLOG functor, except for PROLOG list and functors enclosed by single upper quotes. Logical variables in PROLOG are denoted as character-sequence starting either with a beginning uppercase character (*Age*) or an underscore (*_age*). Atoms in PROLOG, instead, begin with a lowercase (*age*), otherwise an atom’s name must be escaped by surrounding single quotes ‘*Age*’. Class names in JAVA are commonly written in (upper) CamelCase. We convert the CamelCase to snake_case in PROLOG. In order to translate a class name $n_c$, all containing uppercase characters are replaced by their lowercase equivalent. Except for the beginning uppercase character of a class name, replaced characters are given a single underscore character prefix. For instance, the class name *MyClass* in CamelCase is translated to the PROLOG atom *my_class* in snake_case. Then, the functor $t$ as image of $(p_c, n_c)$ under $d$ simply is
the atom resulting from the translations of \( p_c \) and \( n_c \) with a single colon as separator; e. g., \( d(\text{my.new.package, MyClass}) = \text{my.new.package:my.class} \). That way, we guarantee that objects of the same type, but from different packages, form different functors in PROLOG. This is important because terms with the same functor and arity are treated in PROLOG as representatives of the same clause. However, if the connected PROLOG system supports modules, the JAVA package can be resolved to a module following the naming conversion as explained above. For instance in Swi-PROLOG [99], a call for \( \text{my_new_package:my_class} \) is resolved as call to the predicate \( \text{my_class} \) in the module \( \text{my_new_package} \).

**Mapping** \( d(\mathcal{H}_c^l(o) \times \mathcal{F}_c^m(o)) = (a_1^l(o), \ldots, a_n^l(o)) \) of Class Fields. Reference types in JAVA are mapped to compounds in PROLOG whereby the class name is mapped as functor of compound following the rules that we have described above. Every field in \( \mathcal{H}_c^l(o) \times \mathcal{F}_c^m(o) \) is mapped to an argument of the term \( t_c^o \). Therefore, the arity \( n \) of \( t_c^o \) holds the equation \( n = l + m \). Fields of an object \( o_c \) which are not of a primitive JAVA type are mapped under \( d \) recursively. Therefore, the images under \( d \) may result in nested, non-ground terms in PROLOG. However, JAVA arrays are mapped differently. An array is mapped to a plain PROLOG list that contains the array’s elements as members in the same ordering.

Primitive JAVA types of \( \mathcal{P} \) are mapped as follows (to PROLOG): byte (integer), short (integer), int (integer), long (integer), float (float), double (float), boolean (boolean), char (atom). Wrapper classes of primitive JAVA types are translated to PROLOG in the same way as the corresponding primitive. All instances of \texttt{java.lang.String} in JAVA are mapped to the corresponding character-sequence in PROLOG enclosed by single upper quotes, for instance the atom ‘Age’ in PROLOG corresponds to the \texttt{String} instance "Age" in JAVA. If a single upper quote is contained in the JAVA String, it has to be escaped by backslash (\). A reference to \texttt{null} in JAVA is handled particularly in PROLOG, it is always translated to a specific customizable atom. Throughout the rest of the thesis, we translate \texttt{null} in JAVA to the atom \texttt{n_u_l_l} in PROLOG.

Collections and maps in JAVA are mapped to particular PROLOG lists. All instances of \texttt{java.util.Collection} implementations in JAVA; e. g., \texttt{java.util.ArrayList\(<E>\)}, are mapped to plain PROLOG lists. For each element of the collection in JAVA, the list in PROLOG contains a corresponding term as element. The order of the elements of the list in PROLOG is defined by the order in which the elements are returned by iterating the \texttt{Collection} instance in JAVA. All instances of \texttt{java.util.Map\(<K,V>\)} implementations in JAVA, e. g., \texttt{java.util.HashMap}, are mapped to PROLOG lists that contain for each key-value pair a two-element list where each member has the key as first element and the value as second element. For instance, the PROLOG list
[[a, 1], [b, 2]] represents a HashMap instance which maps the two keys a and b to the values 1 and 2, respectively. For convenience, the JAVA-PROLOG type conversions are summarized by Table 5.1.

<table>
<thead>
<tr>
<th>Java</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference type (w/o array)</td>
<td>compound</td>
</tr>
<tr>
<td>array</td>
<td>plain list</td>
</tr>
<tr>
<td>byte (+ wrapper class)</td>
<td>integer</td>
</tr>
<tr>
<td>short (+ wrapper class)</td>
<td>integer</td>
</tr>
<tr>
<td>int (+ wrapper class)</td>
<td>integer</td>
</tr>
<tr>
<td>long (+ wrapper class)</td>
<td>integer</td>
</tr>
<tr>
<td>float (+ wrapper class)</td>
<td>float</td>
</tr>
<tr>
<td>double (+ wrapper class)</td>
<td>float</td>
</tr>
<tr>
<td>boolean (+ wrapper class)</td>
<td>boolean</td>
</tr>
<tr>
<td>char (+ wrapper class)</td>
<td>atom enclosed by single upper quotes</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>atom enclosed by single upper quotes</td>
</tr>
<tr>
<td>null</td>
<td>n_u_l_1</td>
</tr>
<tr>
<td>java.util.Collection</td>
<td>plain list</td>
</tr>
<tr>
<td>java.util.Map&lt;K,V&gt;</td>
<td>list of two-elements lists</td>
</tr>
<tr>
<td>enum</td>
<td>atom</td>
</tr>
</tbody>
</table>

Table 5.1: JAVA-PROLOG type conversion.

5.1.3 Default Mapping Mechanism

In CAPJa, the formal mapping is instantiated as a default mapping mechanism that is applicable to almost any JAVA class. For the mapping process, no modifications to the source code of preexisting classes are necessary. The translation from JAVA to PROLOG and back is internally handled by CAPJa. To translate a JAVA object to a term in PROLOG and to further process the return values, no knowledge in PROLOG is required. In the following, we explain the particularities of the default mapping mechanism of CAPJa.

Regarding the mapping of the class name, the package, and the class fields CAPJa’s default mapping mechanism implements the formalized mapping as described in Section 5.1.2. However, the mapping of class fields in JAVA to ordered arguments of the image in PROLOG requires a special treatment. In JAVA, the fields of a class have no order. Although there are methods in JAVA to retrieve all fields of a class, the order of the returned class fields is not guaranteed and depends on the JVM. Therefore, CAPJa’s default mapping mechanism defines an lexical order based on the names of the class fields. Because field names have to be unique within an individual class,
we obtain a complete order for the class fields. The default mapping mechanism of CAPJa preserves this order during the mapping of class fields to arguments of the image in Prolog.

Table 5.1 reveals that the type conversion from Java to Prolog is not injective because the different Java types are represented by the same Prolog types or lists. However, the default mapping mechanism of CAPJa saves the Java types involved in a query from Java to Prolog and uses this information later on to determine the corresponding Java types for the translation of the return values from Prolog. Therefore, it is also not necessary to map Java type information to Prolog.

For any Java type, the default mapping to Prolog is implemented by a subclasses of the abstract class JPMapper<T> in CAPJa. In the following, we simply refer to a subclass of JPMapper<T> as a mapper. A mapper that implements CAPJa’s default mapping mechanism for a given Java type is also referred as default mapper. Every mapper provides methods that implement the mapping facilities for the type parameter T of the mapper. Each mapper provides methods for the following tasks.

- Represent the type parameter T as Prolog term.
- Represent instances of T as Prolog terms.
- Translate a particular Prolog term into an instance of T.
- Translate particular Prolog variable bindings into an instance of T.

With CAPJa, it is not necessary to manually implement a mapper, its source code can be generated. In the Section 5.3, we clarify how the translation facilities are actually implemented and how the source code of mappers can be generated with the help of CAPJa. The following section explains how CAPJa’s default mapping mechanism can be modified with the help of source code annotations.

### 5.1.4 Custom Mapping Definitions

In Java, CAPJa provides two options for the definition of custom object-to-term mappings. In this section, we describe an annotation-based approach. The second option is subsequently described in Chapter 6 relies on lambda expressions.

For a single Java type, custom mapping definitions can be used to alter the specifications of the image of the mapping to Prolog. These modifications include the functor, arity, argument order, and argument structure of the target in Prolog. For this purpose, the JPMAPPING component of CAPJa provides a compact source code annotation layer. Annotations usually are not part of a Java program and do not affect the source code which they annotate. At runtime, annotations are accessible.
by the methods of Java’s Reflection API [63]; at compile time, they can be evaluated by an annotation processor [61]. For an individual Java class, CAPJa’s annotation layer allows for multiple custom mapping definitions. Multiple mappings can be defined with the help of the three nested annotations @JPMappings, @JPMapping, and @JPFieldMapping. Figure 5.1 illustrates the associated Java interfaces and their dependencies.

The @JPMappings annotation allows us to define multiple custom mappings for an individual class. The annotation has only a single annotation element, an array to list multiple @JPMapping annotations. We only need this annotations in older Java versions, before Java 8. In Java 8, repeated annotations have been introduced and thus the @JPMappings annotation can be omitted.

The @JPMapping annotation is used by CAPJa to generate a mapper that implements the specified custom mapping for the annotated Java type. If necessary, additional custom mappings can be defined by additional @JPMapping annotations. Each @JPMapping annotation specifies the common structure of the term that is the image of the mapping to Prolog. For this purpose, a @JPMapping annotation provides several annotation elements:

- **id** is a mandatory annotation element and is used by CAPJa to identify the mapping. A id has to be unique within the annotated class.

- **isDefaultMapping** marks the custom mapping as default mapping for the annotated class. If CAPJa does not know which mapping should be applied to a given class instance, it always uses the associated default mapper for the translation. Usually a default mapper implements CAPJa’s default mapping mechanism. However, if this annotation element is set true, the default mapper associated with the annotated Java class implements this custom mapping.

- **functor** sets a custom functor. If functor is omitted, the default mapping mechanism applies and translates the class name as functor, instead.

- **mapToList** changes the mapping of class instances from compounds to lists in Prolog. It is not allowed to use this element together with the functor element.
• **argumentOrder** determines which fields are mapped as arguments of the image to PROLOG. As the name of the annotation element suggests, the order of the fields identifiers within the JAVA array determines the order of the resulting arguments in PROLOG.

• **excludeFields** lists class fields that are not mapped to PROLOG. In contrast to **argumentOrder**, remaining fields are translated to arguments in lexical order. The usage of **argumentOrder** and **excludeFields** within the same @JPMapping annotation is not allowed. If **excludeFields** and **argumentOrder** are omitted in a @JPMapping annotation, all class fields are mapped in lexical order to arguments of the image in PROLOG.

• **fieldMappings** lists @JPFieldMapping annotations which further determine the mappings of non-primitive class fields. If **fieldMappings** is omitted in @JPMapping, the default mapping mechanism applies to all mapped class fields.

The @JPMapping annotation can be used to associate a class field with a custom mapping. Suitable class fields are only non-primitive JAVA types, excluding the primitive wrapper classes and the String type. If a class field implements the interface java.util.Collection, then the associated custom mapping applies to all members of the collection. The @JPFieldMapping annotations has the following annotation elements:

• **field** references the identifier of a class field for which we want to specify a custom mapping.

• **mapperId** references a custom mapping via the id of a defining @JPMapping annotation. If the referenced @JPMapping annotation is not in the same source file, the naming convention for the reference is fullyQualifiedName@id, for instance test.Person@1 to reference the defining @JPMapping annotation with the id equals 1 of the Person class of the test package.

• **keyMapperId** and **valueMapperId** have to be used if **field** references a class field that implements the java.util.Map interface. In this way, we define the custom mappings for the keys and values of the map, respectively.

Each annotation is verified by CAPJa before it is evaluated for the generation of a corresponding mapper. Listing 5.4 shows a modified version of the Person class from Section 5.1.3. We have added the class field **children** which lists all children of a Person object. In addition, we have added three custom mapping definitions for instances of the Person class.
The JPMapping Component

```java
@JPMappings({
  @JPMapping(id="1", functor="stimpson",
      argumentOrder={"givenName", "children"},
      fieldMappings={@JPFieldMapping(field="children", mapperId="1")}),
  @JPMapping(id="2", functor="parent",
      fieldMappings={@JPFieldMapping(field="children", mapperId="3")}),
  @JPMapping(id="3", functor="child",
      excludeFields={"children", "familyName"})
})
class Person {
  private ArrayList<Person> children = new ArrayList<>();
  private String givenName;
  private String familyName;
}
```

Listing 5.4: Custom mapping definitions for the Person class in JAVA.

In Listing 5.5, we have instantiated the Person object homer together with three Person instances bart, lisa, and maggie that represent the children of homer.

```java
Person homer = new Person("Homer", "Stimpson");
Person bart = new Person("Bart", "Stimpson");
Person lisa = new Person("Lisa", "Stimpson");
Person maggie = new Person("Maggie", "Stimpson");
homer.setChildren(Arrays.asList(bart, lisa, maggie));
```

Listing 5.5: Person instances for the mapping to PROLOG.

Finally, Listing 5.6 shows the different representations of the homer object in JAVA as term in PROLOG according to CAPJa’s default mapping and the custom mappings defined by the first two @JPMapping annotations of the Person class from Listing 5.4.

```prolog
% person(Children, FamilyName, GivenName)
person([person([], 'Stimpson', 'Bart'), person([], 'Stimpson', 'Lisa'),
        person([], 'Stimpson', 'Maggie')], 'Stimpson', 'Homer').

% stimpson(GivenName, Children)
stimpson('Homer', [stimpson('Bart', []), stimpson('Lisa', []),
                   stimpson('Maggie', [])]).

% parent(Children, FamilyName, GivenName)
parent([child('Bart'), child('Lisa'), child('Maggie')],
       'Stimpson', 'Homer').
```

Listing 5.6: Different representations of the homer object as term in PROLOG.
The `person/3` predicate has been derived with the help of CAPJa’s default mapping mechanism. The `stimpson/2` predicate has been derived from the `@JPMapping` annotation with `id` equals 1. The arguments are not sorted in lexical order but derive form the occurring of the class fields within `argumentOrder`. The `familyName` class field has not been mapped because it was not listed within `argumentOrder`. The `children` class field has been translated as `Prolog list` because it is a subtype of `java.util.Collection` in `Java`. Each member of `children` has been mapped according to the custom mapping definition of the `@JPMapping` annotation with `id` equals 1. The `parent/3` predicate has been derived from the `@JPMapping` annotation with `id` equals 2. Because there is no `argumentOrder` or `excludeFields` annotation element present in the defining `@JPMapping` annotation, all class fields are mapped in lexical order to arguments. The `@JPFieldMapping` annotation within `fieldMappings` defines a custom mapping for the `children` class field which is defined by the `@JPMapping` annotation with `id` equals 3. Each member of `children` therefore is mapped to a term with functor `child` and arity 1. This single argument represents the `givenName` class field.

Our example with the `Person` class already shows the versatility of CAPJa’s mapping mechanism which allows us to easily define different representations of `Java` class instances in `Prolog`.

### 5.2 Predicate-Signature Notation

In Section 5.1.2, we have presented an object-to-term mapping mechanism that can be customized with an annotation layer in `Java`. In this section, we introduce the `Predicate-Signature Notation (PSN)` in `Prolog` which allows `Prolog` programmers to characterize signatures of predicates. A predicate signature describes the composition of predicates with arity greater zero, similar to `XML` Schemata and `XML` documents. CAPJa generates from predicate signatures in `PSN` particular `Java` classes that associate the `Prolog` predicates characterized in `PSN`. The classes generated for `Java` form an object-oriented interface to `Prolog` which can be used to easily query the associated predicates. In this way, CAPJa offers a convenient, semi-automatic predicate integration mechanism.

We begin with an introductory example in Section 5.2.1 that illustrates the usage of the `PSN`. In Section 5.2.2, we explain syntax and semantics of the `PSN` more precisely.
5 The JPMapping Component

5.2.1 Introductory Example

In the following, we want to characterize the predicated signature of a book/6 predicate which has been defined for an exemplary library module in, e.g., SWI-PROLOG. Listing 5.7 shows the book/6 fact for the famous PROLOG book of Ivan Bratko [10].

```
```

Listing 5.7: A book/6 fact in PROLOG.

The third argument of the book/6 predicate is a PROLOG list which represents all authors of a given book. However, in our example, Ivan Bratko is the only author. Lists can have different properties depending on their usage in PROLOG; e.g., they can have fixed sizes or not, be homogeneous, i.e., all list members derive from the same predicate, or not. It is up to the PROLOG developer to characterize a list in PSN according to its usage in the PSN. In our example, the author list may contain one or more author/2 terms and thus has no fixed size. In addition, it is a homogeneous list because list members are only author/2 terms. The predicate signature corresponding to the book/6 predicate in PSN is given by the following listing.

```
%module(+Name, +Operator)
module(library, :).

% predicate(+Name, +Type, +Arguments)
predicate(book/6, compound, [argument(title, atom), argument(isbn, atom),
                             argument(authorList, list, author/2), argument(edition, atom),
                             argument(publisher, atom), argument(year, integer)]).
predicate(author/2, compound, [argument(firstName, atom),
                             argument(lastName, atom)])
```

Listing 5.8: Signature of the author/6 predicate in PSN.

As shown in Listing 5.8, the predicate signature in PSN characterizes the composition of the book/6 predicate in PROLOG. For this purpose, a predicate and its arguments are distinguished and further characterized by particular PROLOG type annotations. As mentioned before, CAPJa generates from predicate signatures in PSN classes in JAVA that associate predicates characterized in PSN. The predicate signatures of our example would lead to the two classes Book and Author in JAVA. For each argument term in PSN, a corresponding JAVA class gets an associated class field. For instance, the argument(authorList, list, author/2) term will be represented by a field of the Book class in JAVA with identifier equals authorList. CAPJa represents
5.2 Predicate-Signature Notation

homogeneous PROLOG lists by the `java.util.ArrayList<T>` type in JAVA. In our example, the type parameter `T` of the `authorList` would be equal to the `Author` type in JAVA. Another example for the usage of the Psn can be found in Section 8.1.1 where we integrate a PROLOG knowledge base into JAVA.

5.2.2 Syntax and Semantics

The signature of predicates is written in plain PROLOG. For the Psn, we define the PROLOG type annotations `atom`, `compound`, `float`, `integer`, and `list` as atoms in PROLOG. A predicate is characterized by

```
predicate(+Name, +Type, +Arguments)
```

in Psn. The `Name` argument of `predicate/3` is specified in the form `functor/arity`. The `Name` argument is used by CAPJa to derive the name of the JAVA class that represents the predicate characterized in Psn. For this purpose, `functor` is used as class name. CAPJa replaces a beginning lowercase character of `functor` by its uppercase equivalent because class names in JAVA commonly begin with a capital. Moreover, if `functor` is in snake_case, the corresponding class name is converted to upper CamelCase. If a class name shall differ from a predicate’s `functor`, `Name` can be alternatively given in the form `functor/arity:className` where `className` is an atom that represents the simple class name. If `className` is enclosed by single quotes, we can also specify a fully qualified class name. If the package associated with the fully qualified class name does not exist, it is created by CAPJa. Depending on the PROLOG type the `Type` argument of `predicate/3` is either the atom `compound` or `list`. The last argument of `predicate/3` is the PROLOG list `Arguments` that describes the arguments of the predicate characterized in Psn. Each arguments is represented by one of the following predicates:

- `argument(+ArgName),`
- `argument(+ArgName, +Type),` or
- `argument(+ArgName, list, +Members).`

The `ArgName` argument is used analogously to `Name` in `predicate/3`. The `argument/1` predicate is used to express very generic arguments such as in the well-known `member(?Elem, ?List)` predicate which proves true if `Elem` is a member of `List`. The class field in JAVA corresponding to an `argument/1` predicate is always of type `java.lang.Object`. Therefore, the most generic signature of `member/2` in Psn is shown in Listing 5.9. For more specific signatures of `member/2` the arity 3 version of `argument` can be used.
The usage of either \texttt{argument/2} or \texttt{argument/3} depends on the value of \texttt{Type} which describes the \textsc{prolog} type of the argument.

- If \texttt{Type} is equal to \texttt{atom}, \texttt{float}, or \texttt{integer}, the \texttt{argument/2} predicate is used.

- If \texttt{Type} is equal to \texttt{compound}, the \texttt{argument/2} predicate is used, too, but in this case we have to define an additional \texttt{predicate/3} term that describes the signature of the actual argument. The \texttt{Name} argument of the additional \texttt{predicate/3} term requires to match the \texttt{ArgName} of the initial \texttt{argument/2} term.

- If \texttt{Type} is equal to \texttt{list}, the \texttt{argument/3} predicate is used and the \texttt{Members} argument characterizes the list members.
  
  - If the associated list is homogeneous with members of the same \textsc{prolog} type \texttt{integer}, \texttt{float}, or \texttt{atom}, the \texttt{Members} argument is the \textsc{prolog} type of the list members.

  - If the list is homogeneous with members of the \textsc{prolog} type \texttt{compound} that have the same predicate signature, the \texttt{Members} argument references their predicate signature by its \texttt{Name}. \textsc{capja} represents \textsc{prolog} lists with homogeneous members by \texttt{ArrayList<T> types in java} which preserve the order of their elements. The type parameter \texttt{T} is \texttt{Integer} for the \textsc{prolog} type \texttt{integer}, \texttt{String} for \texttt{atom}, and \texttt{Float} for \texttt{float}. For compounds the type parameter \texttt{T} associates the \textsc{java} class corresponding to the predicate signature of the given compounds.

  - If the \textsc{prolog} list has members of mixed types, for instance \texttt{compound} mixed with \texttt{atom}, the \texttt{Members} argument references a new \texttt{predicate/3} term via the \texttt{Name} argument that has its \texttt{Type} argument set to \texttt{list} and \texttt{Arguments} describes each member of the original list. Instead of an \texttt{ArrayList}, \textsc{capja} represents non-homogeneous \textsc{prolog} lists by new classes with class fields for each list element. In addition, such a new class requires a \texttt{@JPMapping} annotation with the \texttt{isDefaultMapping} and \texttt{mapToList} annotation elements set \texttt{true}. The \texttt{argumentOrder} annotation element has to ensure that the class fields map accordingly to the list members in \textsc{prolog}.

If a predicate belongs to a \textsc{prolog} module, we use the \texttt{module(+Name, +Operator)} predicate in \textsc{psn} for expressing this affiliation. All predicate signatures following
5.3 Implementation of Mappings

In this section, we describe how mappings between objects in Java and terms in Prolog are actually implemented in CAPJa and how the necessary source code can be generated. We begin with a description of the abstract class JPMapper\langle T\rangle in Section 5.3.1 and illustrate an implementation of JPMapper\langle T\rangle in Section 5.3.2. Finally, we describe in Section 5.3.3 how CAPJa generates the necessary source code that implements an actual mapping.

5.3.1 The Abstract JPMapper\langle T\rangle Class

Mappings between Java and Prolog are generally represented in CAPJa by the abstract generic class JPMapper\langle T\rangle. The definition of the JPMapper\langle T\rangle class is shown in Figure 5.2 and due to lack of space, we have included only the abstract methods of JPMapper\langle T\rangle. The type parameter T of JPMapper\langle T\rangle represents the mapping type in Java. In the following, we describe shortly the role of each abstract method of JPMapper\langle T\rangle in the mapping between Java and Prolog:

- public abstract T getInstanceFromBindings(List\langle String\rangle bindings) returns an object of type T from a list of variable bindings of Prolog in string format.
- public abstract T getInstanceFromTerm(String term) returns an object of type T from an instance of String that represents a term in Prolog.
- public abstract String getTermFromInstance(T t) returns a representation of an object t of type T as Prolog term in string format.

5.3.2 Implementation of JPMapper\langle T\rangle Class

a module/2 term in PSN represent predicates of this module. Because modules are not uniformly treated by available Prolog systems, we have separated the module Name from the Operator used to indicate a call to a particular predicate of a given module. For instance, a call to the custom person/3 predicate of the custom module company is given by company:person(X,Y,Z) in Swi-Prolog.
5.3.2 Subclassing JPMapper<T>

A particular mapping between the instances of a given Java class and a corresponding predicate in PROLOG is implemented in the form of a subclass of JPMapper<T>. Therefore, the abstract methods of JPMapper<T> have to be implemented for the type parameter T which is given by the Java class for which we define the mapping. In the following, we illustrate the implementation of a particular mapping in CAPJa with the help of an example.

In Section 5.1.4 we have defined several custom mappings for instances of the Person class. From the second annotation with id equals 2 of the Person class in Listing 5.4, CAPJa generates the PersonJPCustomMapper2 subclass of JPMapper<T> that implements this particular custom mapping. The generation of subclasses of JPMapper<T> from @JPMapping annotations is described more precisely in the Section 5.3.3. The generated mapper PersonJPCustomMapper2 extends JPMapper<Person> and has Person as mapping type. Class names of generated mappers follow a simple naming convention in CAPJa. The prefix is the corresponding mapping type, in our example Person, followed by either JPDefaultMapper or JPCustomMapper+id. The former is used, if the mapper implements the default mapping for the given mapping type and the latter is used, if the mapper implements a custom mapping defined by a @JPMapping annotation. In our example, the associated @JPMapping annotation has 2 as id.

The implementation of the getInstanceFromBindings method of the Person mapper is shown in Listing 5.10. This method returns a new Person instance from a passed list of PROLOG variable bindings. One after the other, the values for the class fields children, familyName, and givenName are retrieved from passed variable bindings in string format. Because the corresponding @JPMapping annotation of the Person class in Listing 5.10 defines for the children field a mapping according to the third @JPMapping annotation with id equals 3, CAPJa has referenced the mapper corresponding to this annotation. This is an instance of the PersonJPCustomMapper3 class which has been generated by CAPJa, too, and is used to translate the members of the returned PROLOG list into Person objects, see also Line 11 to 13 in Listing 5.10.
Because the subclasses of JPMapper<T> are only used by CAPJas together with queries to PROLOG, arguments of the query goal may have been set to the anonymous PROLOG variable. In addition, fields of the query type in JAVA may have been set to null, which leads to the user-definable atom in PROLOG, throughout this thesis n_u_l_l. In order to handle the anonymous PROLOG variable and null values in JAVA properly, CAPJas uses the anonymousOrNull method which tests if for a given class field either no variable binding has been returned from PROLOG or the binding is equal to n_u_l_l, see also the Lines 8, 15, and 20 in Listing 5.10. In both cases, the respective class field is set to its default value in JAVA; e. g. null for reference types. The class fields firstName and lastName of the Person class are of type String and variable bindings returned from PROLOG can be passed directly as parameter of the corresponding setter methods.

The getInstanceFromTerm method is implemented similarly to the previous method. The only difference is that a PROLOG term in string format is passed as input param-
eter, instead of a list with bindings of PROLOG variables. This term is preprocessed 
by a PROLOG parser which returns a list of variable bindings that are processed in 
the same way as with the getInstanceFromBindings method. The PROLOG parser 
has been implemented with the help of parser generator tool ANTLR [72]. For this 
purpose, we have implemented a PROLOG grammar for ANTLR which is used to gen-
erate the source code of the PROLOG parser in JAVA. Listing 5.11 shows an excerpt 
of the grammar file for ANTLR. The parser rules start with lowercase letters. The 
lexer rules start with uppercase letters and have been omitted in Listing 5.11.

```java
grammar PrologTerm;

term : functor '(', argument (',', argument)* ')'
   | functor;

list : '[' member (',', member)* ']'
   | emptyList;

emptyList: EMPTYLIST;

functor: ATOM;

argument: term
   | list
   | INT
   | constant
   | variable;

member: term
   | list
   | INT
   | constant
   | variable;

constant : ATOM
   | INT;

variable : anonymous
   | VARIABLE;

anonymous: ANONYMOUS;
```

Listing 5.11: PROLOG grammar for ANTLR.

The implementation of the getInstanceFromInstance method of the Person mapper is 
shown in Listing 5.10. This method returns a PROLOG term in string format that
5.3 Implementation of Mappings

represent a passed Person instance according to the custom mapping definition of the @JPMapping annotation with id equals 2 of the Person class of Listing 5.4.

```
public String getTermFromInstance(Person p) {
    if (p == null) {
        return "n_u_l_l";
    }
    return functor + "("
        + getPrologListFromCollection(p.getChildren(),
            new PersonJPCustomMapper3()) + ",",
        + getSimpleArgumentFromString(p.getFamilyName()) + ",",
        + getSimpleArgumentFromString(p.getGivenName())
        + ")";
}
```

Listing 5.12: An implementation of the getTermFromInstance method.

The return of this method is a parent/3 term in string format as defined in Listing 5.6. This term is the PROLOG representation of the input instance p of the Person class. The class variable functor, which in our case is equal to parent, is concatenated with the term representations of the class fields of p in string format. The getPrologListFromCollection method translates JAVA collections to PROLOG lists. According to the @JPMapping annotation with id equals 2, each collection element in JAVA is translated by PersonJPCustomMapper3. The term representations of the class fields are concatenated in the same order as specified by the @JPMapping annotation. The getSimpleArgumentFromString method tests input strings for null values. If such a test is positive, the n_u_l_l atom is concatenated. Class fields that have a primitive JAVA type or primitive wrapper type are simply concatenated in string format. Class fields that have a reference type other than String are translated by the getComplexArgFromReferenceType method. This method has two input parameters: the class field and a mapper instance associated to the reference type. In the case of a JAVA array, the mapper associates the JAVA type of the elements of the array.

The getTerm method is parameterless and returns a PROLOG term in string format, where all class fields are represented by logical variables. It is implemented similarly to the getTermFromInstance method. The implementation of isMappingType method of the PersonJPCustomMapper2 class is shown in Listing 5.13.

```
public boolean isMappingType(Class<?> clazz) {
    return clazz == Person.class;
}
```

Listing 5.13: An implementation of the isMappingType method.

99
The `isMappingType` method returns `true` if the passed `Class` object is of type `Person`. In this way, CAPJa tests the compliance with the mapping type of a given mapper.

### 5.3.3 Source Code Generation

In order to relieve the programmer of the repetitive task of writing *glue-code* for object-to-term mappings, CAPJa supports the generation of source code. Source code generation not only reduces the programming effort but also the number of programming errors. This accelerates the software development significantly. CAPJa provides source code generation in the following two cases:

1. **From Java:** the annotation layer based on `@JPMapping` can be analyzed to generate the associated mappers that implement the custom mappings for the annotated class type.
2. **From Prolog:** predicate signatures in `Psn` can be analyzed to generate an object-oriented interface in the form of `Java` classes that associate the predicates characterized in `Psn`. In the same process, corresponding mapper classes are generated that implement the mapping between the object-oriented interface in `Java` and the predicates characterized in `Psn`.

**From Java.** We have implemented a custom `JAVA` source code generator in `Java` which is called `JPMapperGenerator`. This source code generator analyzes all relevant classes with `@JPMapping` annotations first, then the source files of mapped class fields without `@JPMapping` annotations in order to generate the source code of corresponding mapper, i.e., particular subclasses of `JPMapper<T>`. In the following, we illustrate the generation process with `JPMapperGenerator` with the help of an example.

In Section 5.2.1, we have introduced the `book/6` predicate to represent books as facts in `Prolog`. The following `Book` class together with the `Author` class can be used to represent books alternatively in `Java`. Listing 5.14 shows their implementations in `Java`. The `Book` class has a single `@JPMapping` annotation which defines a custom mapping to `Prolog`. The `argumentOrder` annotation element defines the order of the class fields as arguments of the image in `Prolog`. Because no `functor` annotation element is present, the class name is simply translated to `book` as functor of the image in `Prolog`. The `Author` class has no `@JPMapping` annotation and thus is mapped to `Prolog` according to CAPJa’s default mapping mechanism.
5.3 Implementation of Mappings

Figure 5.3: Generating of subclasses of JPMapper<T> with JPMappingProcessor.

```java
@JPMapping(id="1", argumentOrder={ "title", "isbn", "authorList", "edition", "publisher", "year" })
public class Book {
    private ArrayList<Author> authorList;
    private String edition;
    private String isbn;
    private String publisher;
    private String title;
    private int year;
}

public class Author {
    private String firstName;
    private String lastName;
}
```

Listing 5.14: The classes Book and Author in JAVA.

If we instantiate a Book object with the same properties as the exemplary book/6 fact of Listing 5.7 and apply the mapper associated to the custom mapping which is defined by the @JPMapping annotation of Book, the given Book object would be mapped exactly to the exemplary fact of Listing 5.7. The mapper associated to the @JPMapping annotation of Book has not to be implemented manually, we can use JPMapperGenerator instead. Figure 5.3 illustrates the generation process of the associated subclasses of JPMapper<T>. JPMapperGenerator analyzes first the mapping specifications of all @JPMapping annotations defined in the Book class and then the Author class corresponding to the authors class field of Book. As a result of the generation process with JPMapperGenerator, we get two new JAVA source files. The
BookJPCustomMapper1 class represents the mapper associated to the @JPMapping annotation of Book with id equals 1. The AuthorJPDefaultMapper class implements CAPJa’s default mapping mechanism for the Author class. JPMapperGenerator uses reflection in Java to analyze @JPMapping annotations and the composition of relevant classes.

From Prolog. In Section 5.2 we have introduced the PSN to describe the signature of predicates in PROLOG. JPMapperGenerator also provides a parser component for predicate signatures in PSN which we have called PSNParsers. This parser derives an abstract syntax tree (AST) of each predicate signature which are used by JPMapperGenerator to create the corresponding object-oriented interface in JAVA for the predicates characterized in PSN. The PSN parser again is implemented with the help of the open-source parser generator tool ANTLR [72]. For this purpose, we have designed an ANTLR grammar for predicate signatures in PSN. In addition to PSNParsers, ANTLR generates standard visitor and listener classes in JAVA which allow us to interact conveniently with the elements of derived ASTs. A specialized version (PSNVisitor) of the visitor provided by ANTLR has been implemented to traverse a given PSN-AST and to generate from each predicate/3 fact in PSN a corresponding JAVA class that represents the characterized predicate in PROLOG. In the same process an associated mapper is generated by JPMapperGenerator, too. Mapping specifications implicitly given by the predicate signatures in PSN are translated into corresponding @JPMapping annotations.

If we use JPMapperGenerator for the analysis of the signature of the book/6 predicate which has been specified by Listing 5.8, we get exactly the JAVA classes as previously illustrated in Listing 5.14 that represent the book/6 and author/2 predicates in PROLOG. The only difference to Listing 5.14 is that each class is defined by an individual source file. Figure 5.4 illustrates the interaction of PSNParsers and PSNVisitor with the PROLOG source file book_psn.pl that contains the predicate
signatures in PsN of the book/6 and author/2 predicates in Prolog. In addition to the Java source files of the classes Book and Author, JPMapperGenerator also creates the Java source files of the associated subclasses of JPMapper<T> as we just have discussed in the previous paragraph.
6 The JPLambda Component

Various approaches of the last decade have attempted to solve the issues related to an integration of PROLOG and JAVA. Besides general problems such as portability, applicability and customization, a concise and intuitive query mechanism from JAVA to PROLOG is essential for a seamless integration of PROLOG and JAVA.

In Chapter 3 we have presented query mechanisms, such as the query mechanisms of JPL [89], that leave the translation and implementation of PROLOG data structures to the JAVA programmer. This usually leads to a vast amount of boilerplate code which obscures the actual intention. Moreover, the copying of PROLOG data structures to JAVA requires considerable proficiency with PROLOG. A query mechanism based on strings in JAVA, such as with INTERPROLOG [15] or the PDT Connector library [79], is not object-oriented but error-prone, and programming errors are hard to locate, difficult to identify, and only occur at runtime. A query mechanisms based on an automated JAVA-PROLOG mapping relieves the programmer from repeatedly creating PROLOG wrapper classes in JAVA. However, a fixed conversion between language artifacts in JAVA and PROLOG is often not flexible enough, as we have seen with P@J [21]. A source code annotation layer in JAVA that guides the mapping from JAVA to PROLOG has to be analyzed, which considerably affects the performance at runtime, see also the LOGICOBJECTS approach [17]. If we use a custom annotation processor (AP) [61], we can partially shift the evaluation of annotations from runtime to compile time but only annotated program elements can be accessed in this way. As a result, all relevant JAVA classes have to be annotated which significantly increases the necessary programming effort and excludes JAVA types from the mapping to PROLOG for which the sources files are not available or accessible.

The JPLambda component of CAPJa offers a novel approach that is based on the automated and flexible mapping mechanism of the previously introduced JPMAPPING component. JPLambda provides two domain-specific languages (DSLs) embedded in JAVA that are used to express clear, concise, and object-oriented queries to PROLOG. DSLs have proven very versatile in software engineering and nowadays are omnipresent [31]. For instance, EMBEDDED SQL FOR JAVA is a well-known embedded DSL (eDSL) that has been successfully used to integrate the database query language SQL in JAVA. eDSLs are usually implemented as a program library for the programming language they are embedded in. To bridge the gap between PROLOG
and JAVA, we have adopted the eDSL approach for queries to PROLOG.
The first eDSL is the Java-Prolog Query Language (JpQL). The JpQL allows us to formulate object-oriented queries to PROLOG based on Boolean lambda expressions in JAVA and subgoals with equality and relational operators in JAVA. Individual subgoals can be connected by conditional operators in JAVA. In this way, queries in JpQL are kept clear and concise.

The second eDSL is the Java-Prolog Mapping Language (JpML). The JpML allows us to define custom object-to-term mappings explicitly in JAVA. As an alternative to the @JPMapping annotations, the JpML enables the mapping of those class instances to PROLOG, whose source code is not available or accessible. The JpML provides a factory class for mappers. Similar to the JpQL, the mapping specifications are conveniently expressed with the help of a lambda expression in JpML. Both eDSls are conform with the syntax and semantics of JAVA and commonly compilable. However, we use the eDSls of JPLambda only as specification languages for queries and custom mappings. They have no runtime behavior. The custom source-to-source translator JPCompiler is necessary to process the eDSls of JPLambda. The original JAVA source files with specifications for queries in JpQL or custom mappings in JpML are analyzed and lead to additional, new source files in JAVA. Finally, the original JAVA source files are modified by a reference to the newly generated JAVA source files that efficiently implement the queries to PROLOG or the custom mappings.

The remainder of this chapter is structured as follows. In Section 6.1, we begin with some introductory examples with the JpQL and JpML. In Section 6.2, we describe the query mechanism of CAPJa based on the JpQL and the custom mapping definitions with the JpML in Section 6.3. Finally, the processing of both eDSls by JPCompiler is explained in Section 6.4.

### 6.1 Introductory Examples

We begin with some introductory examples in order to clarify the usage of JpQL and JpML. For this purpose, we consider an excerpt of a larger PROLOG knowledge base for a staff management system whose user interface is implemented in JAVA. In the following, we are interested in facts representing employees of a company. Using CAPJa’s predicate integration mechanism based on the Psn, see also Chapter 5, the employee facts in PROLOG have been represented by instances of the classes Employee and Address in JAVA. Fig. 6.1 illustrates both JAVA types and their dependency. The Employee class has the properties first name, last name, salary, and address. Except for address, all other properties are primitive JAVA types.
6.1 Introductory Examples

Now, we can use CAPJa’s query mechanism based on the JPQL to query the PROLOG knowledge base. Suppose we would like to query for Employee instances that have a monthly salary greater than 5000 euros and Baker as first name or last name. In order to speed up the processing of the query and to minimize the data exchanged between PROLOG and JAVA, we want to omit the address property of returned Employee instances. Listing 6.1 shows the corresponding query in JPQL where we have created an instance of the JPQuery<T> class with Employee as query type.

```
JPQuery<Employee> query = new JPQuery<Employee>(
    employee -> employee.getSalary() > 5000
    && ( employee.getFirstName() == "Baker"
    || employee.getLastName() == "Baker" )
    && JPConstraint.omit(employee.getAddress()));
```

Listing 6.1: Querying PROLOG with JPLAMBDA.

The lambda expression provides an employee template of the employee class which can be used to specify the query constraints regarding the salary, first name, last name, and the omit instruction for the address property. All query constraints are based on simple JAVA operators (==, >) or the static methods of the JPContsraint class of JPQL. Individual query constraints are connected by conditional JAVA operators (&&, ||). In this way, queries in JPQL are completely object-oriented and thus natural to JAVA programmers. In addition, the JPQL is very clear and concise. Even for complex queries, the necessary programming effort is reduced to an absolute minimum. If we apply CAPJa’s default mapping, then Listing 6.2 shows the query in PROLOG corresponding to the query in JAVA of Listing 6.1.

```
?- ( employee(_, 'Baker', LastName, Salary)
    ; employee(_, FirstName, 'Baker', Salary), FirstName \= 'Baker' ),
    Salary > 5000.
```

Listing 6.2: Corresponding query in PROLOG.
The first goal is the default representation of Employee as term with a single anonymous variable followed by the query constraints defined in query. In the next Section 6.2, we will explain the options of JPQL more detailed.

Suppose we want to assert German employees from JAVA as compounds to the knowledge base in PROLOG with functors in German. Then we have to define a suitable mapper in JAVA that implements this specific object-to-term mapping. For this purpose, we can use the JPML. Compared to the annotation layer based on @JPMapping, the advantage of the JPML is that, even if the source code of the Employee type is not available or should not be modified by source code annotations, we may define a desired object-to-term mapping in JAVA. Listing 6.3 shows the creation of a mapper in JPML that implements a custom mapping for instances of the Employee class.

```
JPMapper<Employee> employeeMapper = new JPMapperFactory<Employee>().create("1",
(employee, employeeTarget) -> {
    employeeTarget.setFunctor("angestellter");
    employeeTarget.setArgumentOrder(employee.getFirstName(),
    employee.getLastName(), employee.getAddress());
    employeeTarget.setArgumentMapping(employee.getAddress(),
    (address, addressTarget) -> {
        addressTarget.setFunctor("adresse");
        addressTarget.setArgumentOrder(address.getStreet(),
        address.getCity(), "Deutschland");
    });
});
```

Listing 6.3: Instantiating a custom mapper for the Employee class.

The return of the create method of the JPMapperFactory class is an instance of the JPMapper<Employee> class with Employee as mapping type. Similar to the JPQL, the JPML relies on lambda expressions. In JPML, we use the lambda expression to specify the custom object-to-term mapping for the mapping type. The lambda expression provides two templates, one template for the Employee mapping type in JAVA and the other template for the target term in PROLOG. A German Employee instance with first name, last name, street, and city equals Max, Mustermann, Spandauer, and Berlin, respectively, is then mapped to PROLOG by employeeMapper as the term shown in Listing 6.4.

```
angestellter('Max','Mustermann', adresse('Spandauer', 'Berlin', 'Germany'))
```

Listing 6.4: Exemplary target term in PROLOG for employeeMapper.

According to the lambda expression given in Listing 6.3, the functor of the target in PROLOG is equal to angestellter (German word for employee) and the argument representing the address has adresse as functor. In Section 6.3, we will discuss the options of JPQL more detailed.
6.2 The Query Language JPQL

To formulate complex queries to PROLOG in a clear and concise manner, CAPJJa provides the Java-Prolog Query Language (JPQL) which consists of the following JAVA types:

- The generic JPQuery<T> class to define a query with query type T.
- The functional interface JPTester<T> to define query constraints.
- The JPConstraint class to express particular query constraints associated with PROLOG.

In order to state a query to PROLOG with JPQL, we first need to create an instance of the generic class JPQuery<T>. The type parameter T determines the query type, such as Employee, if we query for instances of the Employee class. If we do not specify a mapper for the mapping type of a query, CAPJJa automatically applies the default object-to-term mapping mechanism as described in Section 5.1.3. If we are only interested in arbitrary instances of the query type without any query constraints, we simply can instantiate JPQuery<T> with a parameterless constructor. However, if we want to add query constraints, we have to use the single parameter constructor of JPQuery<T>. This constructor parameter is the functional, generic interface JPTester<T> that has the query type T of JPQuery<T> as type parameter.

6.2.1 Lambda Expressions

A functional interface in JAVA is an interface that only defines a single abstract method which has to be implemented in order to satisfy the requirements of the interface. The single method of JPTester<T> is the Boolean method test(T t). Because the constructor of JPQuery<T> has a functional interface as parameter, we can pass a lambda expression in order to implement the test method of JPTester<T>. In JAVA, lambda expressions are the answer for functional objects and enable code as data. Generally, lambda expressions, also referred to as anonymous functions, in programming are defined as functions or subroutines that can be called without being bound to an identifier, which often leads to a lighter syntax as compared to common methods or named functions. Anonymous functions are ubiquitous in functional programming languages and originate from lambda calculus, which was invented by Alonzo Church in the 1930s prior to electronic computers. In lambda calculus all functions are anonymous. Church’s formal definition for lambda expressions reads as follows.

A lambda expression is composed of
• variables $v_1, v_2, \ldots, v_n, \ldots$,
• the abstraction symbols lambda $\lambda$,
• the dot symbol, and
• round brackets.

The set of all lambda expressions, $\Lambda$, can be defined inductively.

1. If $x$ is a variable, then $x \in \Lambda$.
2. If $x$ is a variable and $M \in \Lambda$, then $(\lambda x. M) \in \Lambda$.
3. If $M, N \in \Lambda$, then $(M N) \in \Lambda$.

Instances of rule 2 are commonly referred to as *abstractions* and instances of rule 3 are known as *applications*. An example for a lambda is $((\lambda x. 2) (\lambda x. x + 1))$. The expression $(\lambda x. 2)$ can simply be rewritten as function $x \mapsto 2$ that maps each $x$ to the constant 2. Then, the given lambda is obviously equivalent to $x \mapsto 2 + 1$. If we modify the given lambda to $(\lambda x. \lambda y. x + y)$, we get the more general function $(x, y) \mapsto x + y$ of the two variables $x, y$. In JAVA, lambda expressions are denoted in a similar fashion.

```
(int x, int y) -> x+y
```

Listing 6.5: Simple lambda expression in JAVA.

In JAVA, the syntax of lambda expressions is a parenthesized list of arguments followed by the arrow token ($\rightarrow$) and concluded by a body that is either a single expression or a block more statements between balanced braces. If there is only a single argument before the arrow token, the parentheses can be omitted. Argument types in a lambda can be omitted if the target types have been previously declared. The JAVA compiler infers the type parameters and thus explicit retyping is not necessary for the arguments of the lambda expression. Any lambda expressions can be simply substituted by anonymous inner classes in JAVA. However, anonymous classes are much more bulky [65] and on the bytecode level lambda expressions rely on a special *invokedynamic* instruction which can improve the runtime performance.

### 6.2.2 Query Constraints

The JPQL uses lambda expressions to specify query constraints that are associated with the query type and that each result returned from PROLOG has to satisfy. The lambda expression of JPQL start with a template that represents the query type. After the arrow token, individual query constraints can be formulated as Boolean expressions that refer the to query type or its properties.
6.2 The Query Language JPQL

Listing 6.6: Simple lambda in JPQL with a single query constraint.

```java
(Employee employee) -> employee.getFirstName() == "Peter"
```

Because the `test` method of `JPTester<T>` is a Boolean method, each query constraint has to be formulated as a Boolean expressions in JAVA. Actually, the JPQL supports the following Boolean expressions for query constraints:

- equations with equality or relational operators (==,>,>=,<,<=),
- string comparison with the `equals` method of `java.lang.String`, and
- query constraints specific to PROLOG using static methods of the `JPConstraint` class.

One operand always has to be associated with a property of the query type. The JPQL only accepts references to public class fields or implicit references via getter methods. Constraints supported by JPQL are translated to variable bindings or corresponding subgoals in PROLOG of the actual query. In this way, the query constraints are propagated from JAVA to PROLOG and the number of solutions returned from PROLOG is often reduced significantly which reduces the data exchanged between PROLOG and JAVA. A query constraint based on the comparative operator `==` or the `equals` method of the `String` class leads to a substitution of the logical variable corresponding to a property of the query type. For instance, the query constraint in Listing 6.6 substitutes the PROLOG variable that represents the `firstName` property of the `Employee` type by the atom "Peter" in the query to PROLOG. A query constraint based on a relational operator defines a subgoal in PROLOG for a logical variable that represents a property of the query type. For instance, the first query constraint for the `salary` property of the `Employee` type in Listing 6.1 leads to the subgoal `(Salary > 5000)` in PROLOG. Query constraints not supported by JPQL have to be verified subsequently in JAVA. In Section 6.4, we clarify the splitting into expressions for the evaluation in PROLOG or in JAVA. For more complex queries, multiple query constraints can also be combined using the conditional operators `&&` and `||` in JAVA. The `&&` and `||` operators are conditional AND and OR operations on two boolean expressions and both operators exhibit a `short-circuiting` behavior, i.e. the second operand is only evaluated if the first operand is true. This is similar to the evaluation behavior in PROLOG. If a subgoal in PROLOG fails, subsequent goals are not evaluated; instead, backtracking occurs (if possible). Precedence between (combined) query constraints can be easily denoted by round brackets, just like in PROLOG.

Some concepts from PROLOG that do not exist in JAVA still can be expressed in JPQL with the help of the `JPConstraint` class. The usage of the methods of `JPConstraint`
is advanced and requires some knowledge in PROLOG. The generic static Boolean method `unify(S s, T t)` of `JPConstraint` tests if the terms that represent the passed objects `s` and `t` unify in PROLOG. For both objects, CAPJa’s default mapping mechanism applies in order to unify their representations as PROLOG terms. In Section 6.3, an extended version of the `unify` method is described that allows us to pass additional mappers as method parameters that define the desired mappings explicitly for `s` and `t`. The usage of the generic static Boolean method `omit(S s)` of `JPConstraint` can be used to express a lack of interest for a specific property of the query type to speed up the evaluation and to reduce the amount of processed data. The `omit` method of `JPConstraint` translates a property of the query type referenced by `s` into an anonymous variable in PROLOG. The anonymous variable is denoted by a single underscore sign in PROLOG and unifies with every given term. Because PROLOG does not return a binding for the anonymous variable, the property for which the anonymous variable has been created is set to its default value (usually null for non-primitive JAVA types) in returned instances of the query type. For instance, the invocation of the `omit` method of the example in Listing 6.1 leads to `Employee` instances with address property equals null in JAVA.

To avoid duplicates in the set of matching data in PROLOG due to backtracking, we translate conditional OR operations in JAVA such as `Condition1 || Condition2` to

```
Condition1 ; (Condition2, \=(Condition1))
```

in PROLOG. For instance, in the exemplary query of Listing 6.1 we have queried for instances of `Employee` instance with first or last name Baker. Suppose there is an employee with first name and also last name Baker once in the PROLOG knowledge base. If we translate conditional OR operations in JAVA to standard disjunctions in PROLOG, we would get two `Employee` instances with first name and last name Baker. PROLOG would test first for employees with first name equals Baker and then apply backtracking to test for employees with last name equals Baker.

### 6.2.3 Query Execution

The following four methods of `JPQuery<T>` can be used to execute the query via a given gateway that abstracts the connectivity with an individual PROLOG system and its internal database:

- `boolean hasSolution(JPGateway gateway)`,
- `T getSolution(JPGateway gateway)`,
- `List<T> getAllSolutions(JPGateway gateway)` and
6.2 The Query Language JPQL

- `Iterator<T> getLazySolutionsIterator(JPGateway gateway)`.

The `hasSolution` method asks PROLOG for the mere existence of a solution to a query from JAVA. In the case of existence, `hasSolution` returns `true` in JAVA, otherwise `false`. If we are interested only in the variable bindings of the first solution in PROLOG, we simply call `getSolution`. The `getSolution` method implicitly prevents backtracking in PROLOG. If we want all possible solutions of a query to PROLOG at once, we call the `getAllSolutions` method that returns a list with all solutions. This call is deterministic and uses internally the meta-predicate `findall(+Template, :Goal, -List)` in PROLOG. Members of List which unify with Template and are inferred from the solutions of Goal via continuous backtracking. The method `getLazySolutionsIterator` of `JPQuery<T>` provides control over backtracking in PROLOG. Its return value is a specialized `Iterator` instance [34] which allows us to traverse all solutions using the `hasNext` and `next` methods of `Iterator`. The results of the query to PROLOG are not cached in JAVA; each invocation of `hasNext` requests PROLOG for backtracking. The `next` method then returns the actual result obtained by backtracking in PROLOG. Listing 6.7 shows the usage of all three methods of `JPQuery<T>`. The methods to query PROLOG can be invoked also without a JPGateway reference. In that case, CAPJa automatically addresses a gateway instance which has been globally designated as `current` gateway.

```java
Employee firstEmployeeFound = query.getSolution(gateway);
List<Employee> employeeList = query.getAllSolutions(gateway);
Iterator<Employee> it = query.getLazySolutionsIterator(gateway);
while(it.hasNext()){ // traverse the set of solutions with...
    Employee e = it.next(); //...lazy evaluation via backtracking
    // process Employee e further...
}
```

Listing 6.7: Executing a query to PROLOG in JPQL.

Queries with query constraints that are not supported by JPQL include a subsequent testing in JAVA for any solutions returned from PROLOG. In this way, JPQL guarantees that a solution satisfies all query constraints stated within the lambda expression of a query. If a solution from PROLOG does not satisfy the tests in JAVA, it is discarded and PROLOG is requested for backtracking. This procedure continues until a valid solution is obtained or all solutions in PROLOG are exhausted. If no solution can be found, the `hasSolution` method returns `false`, the methods `getSolution` and `getAllSolution` return both `null`, and the `getLazySolutionIterator` method returns an empty iterator.
6.3 The Mapping Language JPML

If a given class in Java and the image of the mapping to Prolog do not comply with CAPJa’s default mapping, an object-to-term conversion can be customized. We have already described an annotation-based approach in Section 5.1.4 to define custom object-to-term mappings. However, if the class for which we want to define a custom object-to-term mapping can not be annotated because its source code is not available or not accessible, we need an alternative. We approach this issue with the Java-Prolog Mapping Language (JPML) in a similar manner as queries to Prolog in JPQL. The advantage of the approach with JPML is that the source code of Java types we want to map to Prolog requires no annotations with @JPMapping. The JPML is a compact eDSL in Java which allows us to define custom mappings explicitly in Java. It consists of the following types:

- the generic factory class JPMapperFactory<T> to create a mapper instance,
- the functional interface JPMapping<T> to define a custom object-to-term mapping,
- the JPTarget class to characterize the target of the mapping in Prolog, and
- the JPAnonymous class to indicate an anonymous variable in Prolog.

A custom mapper can easily be defined in JPML with the help of the method

\[
JPMapper<T> create(String id, JPMapping<T> mapping)
\]

of the JPMapperFactory<T> class with the type parameter T that defines the mapping type of the new mapper instance. The first method parameter is used to derive the class name of the JPMapper<T> subclass. The class name consists of the mapping type as prefix followed by JPCustomMapper+id; e.g., the resulting class name of the mapper defined in Listing 6.3 is EmployeeJPCustomMapper1. If CAPJa already has registered a mapper with an identical class name as defined by an invocation of the create method, an exception is thrown. The second method parameter mapping is the functional interface JPMapping<T> and we use a lambda expression to implement its only method setObjectToTermMapping(T object, JPTarget target) which specifies the custom object-to-term mapping. The lambda expression of JPML has the following form:

\[
\{(object, target) -> { /* characterize target in Prolog*/ }\}
\]

The first argument object of the lambda expression is a template for the mapping type T. The second argument target is of type JPTarget and characterizes the target of the mapping to Prolog. The JPTarget class provides the following methods to characterize the composition of the target in Prolog:
6.3 The Mapping Language JPML

1. `void setFunctor(String functor),`
2. `void setArgumentOrder(Object... obj),`
3. `void excludeFields(Object... obj), and`
4. `void setArgumentMapping(T field, JPMapping<T> mapping).`

The `setFunctor` method defines the functor of the target in PROLOG. The parameters of the `setArgumentOrder` method determine which class fields of `object` are mapped to arguments of the target in PROLOG. The order of the method parameters defines also the order of corresponding arguments in PROLOG. Only public class fields or references via getter methods of `object` are valid method parameters. Using an instance of `JPAnonymous` as method parameter in `setArgumentOrder` leads to an anonymous variable as argument at the corresponding position of the target in PROLOG. For convenience, the `excludeFields` method can be used as an alternative to `setArgumentOrder`. The parameters of `excludeFields` simply determine which class fields of `object` are not mapped to PROLOG. The remaining class fields are sorted as arguments of the target in PROLOG with lexical order. The usage of the `excludeFields` method together with the `setArgumentOrder` method leads to an exception. The generic `setArgumentMapping` method defines a custom mapping for a given `field` of `object` that is neither of type `String` nor a primitive JAVA type (or primitive wrapper type). For instance, the `address` field of the `Employee` class is a reference type for which we have defined in Listing 6.3 a custom mapping. The first method parameter `field` of `setArgumentMapping` is a reference to an class field of `object` for which we want to define the custom mapping. The second parameter `mapping` is used in the same way as the identical method parameter of the `create` method.

To control the mapping of the query type of a query to PROLOG, a mapper instance can be passed as additional first parameter to the constructor of `JPQuery<T>`. By default, the query type is mapped to PROLOG according to CAPJa’s default mapping mechanism. If we pass a custom mapper as additional constructor parameter, the custom mapper is responsible for the mapping of the query type. However, the mapping type of mapper has to match the query type of the `JQuery<T>` instance. Due to method overloading in JAVA, the JPQL provides an additional version of the static Boolean method `unify` of the `JPConstraint` class with additional method parameters for custom mappers:

```java
boolean unify(JPMapper<S> sMapping, S s, JPMapper<T> tMapping, T t).
```

The two-parameter version of `unify` as described in Section 6.2 applies CAPJa’s default mapping mechanism to `s` and `t`. The four-parameter version of `unify` has
6 The JPLambda Component

Figure 6.2: Extended build process with JPCompiler.

the two additional parameters sMapping and tMapping which define custom object-to-term mappings for s and t.

6.4 JPCompiler

For a seamless integration of Prolog and Java, the JPLambda component of CAPJa provides eDSLs in Java for queries and custom mappings to Prolog. For both eDSLs, CAPJa relies on lambda expressions in Java. However, the eDSLs of JPMAPPING only serve as specification languages for query constraints and custom mappings definitions. Therefore, the used lambda expressions have no runtime behavior and thus are not evaluated at runtime in Java. The intended runtime behavior is derived and implemented during an extended build process based on the custom source-to-source translator JPCompiler that works on top of a standard Java compiler, e.g., javac. Figure 6.2 illustrates the extended build process with JPCompiler. JPCompiler generates and links the new source files to the original source files with lambda expressions of JPQL and JPML. The extended build process with JPCompiler does not create any compatibility problems, the standard Java compiler remains completely unaffected. Although dysfunctional, original source code implemented with CAPJa compiles without errors. The extended build process has only to be applied once, just before a program developed with CAPJa is executed for the first time. Therefore, the development can be completed before the extended build process has to be applied.

JPCompiler analyzes lambda expressions in JPQL or JPML and generates specific Java source code that implements the contained specifications for query constraints and custom mappings. At runtime, the source code generated by JPCompiler is evaluated to execute a query to PROLOG or map an object in Java to a term in PROLOG. For queries with JPQL, the generated source file defines a subclass of the generic abstract JPQueryTranslator<T> class. For custom mappings with JPML, the generated source file defines a custom mapper, i.e., a subclass of the abstract JPMapper<T> class. The generated source files have to be referenced in the original source files that contain the lambda expressions of JPQL or JPML. For this purpose, JPCompiler modifies the original source files slightly. For queries with JPQL, JPCompiler adds an instance of the generated subclass of JPQueryTranslator<T>
as second constructor parameter of JPQuery<T>. For the creation of custom mapper instances with JpML, JPCompiler adds an instance of the generated JPMapper<T> subclass as second method parameter of create. Figure 6.3 illustrates the workflow of JPCompiler, which we will discuss with greater detail in the following paragraph.

### 6.4.1 Workflow

JPCompiler is executed after a standard Java compiler has finished the compilation of all available Java source files. The operations of JPCompiler can be roughly divided into four subsequent steps which are also marked at the right border of Figure 6.3:

1. **Java source files** that contain the relevant lambda expressions are parsed by JPCompiler and an abstract syntax tree (AST) for each lambda is derived. For the parsing, JPCompiler uses the Java8Parser class.

2. With the help of the ParseTreeWalker class the derived AST is traversed. ParseTreeWalker has the two listener attached that are instances of the JPMapperListener and JPQueryListener classes. Based on the observer design pattern, the listener instances react on the event that the walker passes a specific node of the AST.

3. If the ParseTreeWalker passes a node that represent class instance creations of JPMapperFactory<T> or JPQuery<T>, both listener extract the subtree with the lambda expression and analyze it with the help of the JPLambdaAnalyzer class that can be considered the heart of JPCompiler. The JPLambdaAnalyzer class contains the transformation logic for both eDSLs in JPMAPPING. With the help of JPLambdaAnalyzer, the lambda expression of a query to PROLOG is split into elements that can be translated by JPLambdaAnalyzer to PROLOG and Java expressions that cannot be translated to PROLOG. The source code of the latter remains unmodified by JPLambdaAnalyzer. However, Java expressions not translated to PROLOG are subsequently executed to validate the results returned from PROLOG.

4. From the analysis results, JPLambdaAnalyzer generates new Java source code which implements for each processed node of the AST either a subclass of JPMapper<T> or a subclass of JPQueryTranslator<T>. These subclasses are usually created as new Java source files. However, in some cases they have to be added to the analyzed source files in the form of additional inner classes. This depends on references used in the lambda expression. If a reference to a private inner class or to one of its class fields is used within the lambda
Figure 6.3: Workflow of JPLambda Component.
expression, then the derived source code has to be appended as additional inner class. The reason for this is that a private inner class or one of its class fields can only be referenced within the outer class.

### 6.4.2 Source Code Parsing

The parser **Java8Parser** has been generated with the help of the parser generator tool **ANTLR** [71] which is written in **Java** but not limited to **Java**. It can also be used to generate parsers for C++, Python, or any other language. The following remarks on **ANTLR** and a complete introduction to the used techniques can be found in [72].

ANTLR uses a new parsing technology called Adaptive LL(*) or ALL(*) that Terrence Parr developed together with Sam Harwell. ALL(*) is an extension to the former version LL(*) that has been used in ANTLR version 3. It performs grammar analysis dynamically at runtime rather than statically, before the generated parser is executed. Because ALL(*) parsers have access to actual input sequences, they can always figure out how to recognize the sequences by weaving through the grammar appropriately. Static analysis, on the other hand, has to consider all possible (infinitely long) input sequences. Compared to regular expressions, ANTLR grammars are stronger. Regular expressions cannot be used to recognize initializations because of nested initializers. Moreover, they have no memory in the sense that they cannot remember what they have matched earlier. For instance, regular expressions do not know how to match up left and right curly brackets. ANTLR in version 4 automatically rewrites left-recursive rules in a grammar into non-left-recursive equivalents. The only constraint is that the left recursion must be direct, i.e. rules immediately reference themselves. Rules cannot reference another rule on the left side of an alternative that eventually comes back to reference the original rule without matching a token. A parser generated by ANTLR automatically creates convenient representations of the input in the form of parse trees. The parse trees then can be traversed to trigger events that are immediately handled if the current node of the parse tree is of interest. For instance, a tree walker can fire callbacks to a listener. For CAPJAs, we have used a grammar [73] that characterizes the latest version (8) of the **Java** programming language to generate a **Java** parser. Next to a parser, ANTLR generates basic listener classes that provide a set of empty default implementations which we have specialized for our purposes. The classes **JPMapperListener** and **JPQueryListener** used by **JPCompiler** are these specialized listeners.
6.4.3 The JPLambdaAnalyzer Class

The JPLambdaAnalyzer class is the heart of JPCompiler and counts over 3000 source lines of code in JAVA. JPCompiler analyzes each node of parse subtrees that represents lambda expression in instances of JPMapperFactory<T> or JPQuery<T>. For this, JPLambdaAnalyzer traverses the subtree in depth-first manner from left to right while analyzing each node encountered. JPLambdaAnalyzer has different methods specific to the supported expressions in JPML and JPQL. These methods handle either the current node or the subtree of the child nodes of the current node.

In Section 6.2 we have described the kind of query constraints currently supported by JPQL. In order to extend the supported JAVA expressions of JPQL, additional methods can be added to JPLambdaAnalyzer that handle additional JAVA expressions. Provided that there is a predicate in PROLOG that has the same functionality as a given method in JAVA, a translation can applied, every time JPLambdaAnalyzer encounters a node in the passed subtree that represents the given method invocation. In this way, even new types of query constraints can be added to JPQL. For instance, the method contains(String sub) of the java.lang.String class tests if a given String instance contains the sub character-sequence. In SWI-PROLOG, the substring/5 predicate can be used to accomplish the same. Therefore, we can implement a fixed mapping of the contains method in JAVA to the substring/5 predicate in PROLOG. However, it should be noted that not all PROLOG implementations provide the substring/5 predicate as it is not part of the PROLOG ISO-Standard. Such extensions to JPLambdaAnalyzer lead to specific version of CAPJa and affect the portability of resulting programs. On the other side, such extensions may lead to programs that are even more efficient and concise.

The analysis of the create method of the JPMapperFactory<T> class is much easier than the analysis of instantiations of the JPQuery<T> class because the supported expressions within the lambdas of JPML are limited. JPLambdaAnalyzer only handles the nodes or subtrees corresponding to valid expression, other expressions are ignored. The analysis of lambda expressions for instantiations of the JPQuery<T> class is more complex. JPLambdaAnalyzer requires more particular methods to handle the greater variety of occurring nodes and subtrees. For instance, JPLambdaAnalyzer provides methods to handle various expressions such as method invocations, class instance creations, literals, equality and relational expressions, and conditional expressions. Subtrees that represent query constraints not supported by JPQL are marked that they cannot be translated to PROLOG. For instance, a comparison of reference types with the help of the equals method cannot be translated to PROLOG because that would mean that we have to translate individual implementations of equals to PROLOG. The problem with nodes and subtrees that cannot be mapped
to PROLOG is that then, in certain circumstances, predecessor nodes cannot be mapped to PROLOG, too. For instance, if the left hand side of an equation can be mapped to PROLOG, but the right hand side of an equation subsequently cannot be mapped to PROLOG, the entire subtree representing the equation cannot be mapped to PROLOG.

Once the complete AST of a particular lambda expression has been traversed, JPLambdaAnalyzer processes the collected analysis results. For each invocation of the create method of JPMapperFactory<T>, JPLambdaAnalyzer generates similarly to JPMapperGenerator for @JPMapping annotations and predicate signatures in PSN a subclass of JPMapper<T> that implements the custom mapping specified by the lambda expression for the create method. For each instantiation of JPQuery<T>, JPLambdaAnalyzer generates a subclass of the abstract JPQueryTranslator<T> class which will discuss in the course of the following paragraph.

### 6.4.4 Generated Source Code

The source code generated by JPLambdaAnalyzer defines subclasses either of the abstract JPMapper<T> class or the abstract JPQueryTranslator<T> class. In Section 5.3.1, we already have described the abstract JPMapper<T> class and presented an exemplary implementation in Section 5.3.2. In the following, we introduce the abstract JPQueryTranslator<T> class and its subclasses that implement query constraints of given instantiations of JPQuery<T>. Next to utility methods, the abstract JPQueryTranslator<T> class has the following abstract methods which have to be implemented by JPLambdaAnalyzer from the analysis results:

- abstract void setVariableBindings(),
- abstract String getSubgoals(), and
- abstract boolean validate(T t).

The setVariableBindings method is not used to get variable bindings from PROLOG, instead it is used to register query constraints based on the comparative operator == in JAVA. A query without query constraints leads to a PROLOG goal that has the translated class name of the query type as functor and class fields mapped of the query type are translated to individual variables in PROLOG. Suppose we add now a query constraint based on the comparative operator == for a mapped primitive class field of the query type, then this class field is not translated to a PROLOG variable, instead it is substituted by the value of the other operand of the equation. Such assignments are implemented by the setVariableBindings method of
JPQueryTranslator<T>. All other query constraints that can be translated to PROLOG form subgoals that are associated with the called predicate corresponding to the query type in JAVA. The getSubgoals methods implements all subgoals that have been derived by JPLambdaAnalyzer from query constraints in Jpql. Query constraints that cannot be translated to PROLOG are implemented by the validate method of JPQueryTranslator<T>. This method is executed subsequently in JAVA on top of the results returned by PROLOG. Only results from PROLOG that have been tested positive by validate are considered valid solutions of a query to PROLOG. If all query constraints have been translated to PROLOG, the validate method simply returns true. In the following, we discuss the implementations of the abstract methods of JPQueryTranslator<T> that have been derived by JPCompiler from the query shown in Listing 6.8.

```
1  JPQuery<Employee> q1 = new JPQuery<Employee>(
2       employee -> employee.getFirstName() == "Peter"
3           && employee.getSalary() > 3000
4           && employee.getLastName().contains("ar"));
```

Listing 6.8: A query with one query constraint not supported by JPQL.

The JPQuery<T> instance q1 has the Employee type as query type and three query constraints. We query for employees that have Peter as first name, a salary greater than 3000 euros, and a last name that contains the character-sequence ar such as in Parker. Without the query constraints, the Employee query type would be mapped according to its definition in Figure 6.1 and CAPJa’s default mapping mechanism to the PROLOG goal employee(A, F, L, S). The first query constraint is based on the comparative operator == and assigns the PROLOG variable F representing the firstName property of the Employee class to the value Peter. The second query constraint is an inequality and thus is translated to the subgoal (S > 3000) in PROLOG where S represents the salary property of the Employee class. The last query constraint is based on a JAVA expression with the Boolean contains method of the java.lang.String class and is not supported by JPQL. Therefore, the third query constraint cannot be mapped to PROLOG.

The implementations derived by JPLambdaAnalyzer for the abstract methods of JPQueryTranslator<T> are shown in Listing 6.9. The setVariableBindings method informs the actual mapper of the query type about the variable bindings that result from the first query constraint based on the comparative operator == in JAVA. The getSubgoals methods returns the PROLOG subgoal corresponding to the query constraint about the salary property in string format. Because the last query constraint of q1 cannot be translated to PROLOG, the validate method implements the third query constraint unmodified.
protected void setVariableBindings() {
    getMapper().getVariableBindingMap().put(".getFirstName()", "Peter");
}

protected String getSubgoals() {
    return getMapper().getAttributeVariableMap().get("salary")
    + " > 3000";
}

protected boolean validate(Employee employee) {
    return employee.getLastName().contains("ar");
}

Listing 6.9: Implementations of the abstract methods of JPQueryTranslator<T>.

6.4.5 Modified Source Code

In order to provide custom mapping definitions in JPML or queries in JPQL with runtime behavior, the original JAVA source files with the corresponding expressions in JPML and JPQL have to link instances of the associated classes that have been derived and generated by JPLambdaAnalyzer. For this purpose, the original source files are slightly modified by JPLambdaAnalyzer to inject the necessary references. For this purpose, the JPMappeFactory<T> class has an additional version of the create method with a second method parameter for a JPMapper<T> instance. JPLambdaAnalyzer modifies the method invocation of create to the two-parameter version with an instance of just generated subclass of JPMapper<T> as second parameter. Likewise, the JPQuery<T> class provides additional constructors that all have an additional last parameter of type JPQueryTranslator<T>. Listing 6.10 shows how JPLambdaAnalyzer modifies the original source file with the q1 instance of JPQuery<T> from Listing 6.8.

```java
JPQuery<Employee> q1 = new JPQuery<Employee>()
    employee -> employee.getFirstName() == "Peter"
    && employee.getSalary() > 3000
    && employee.getLastName().contains("ar"),
    new JPQueryTranslator1()); // injected reference
```

Listing 6.10: Injecting a reference to the generated JPQueryTranslator1 subclass.

JPLambdaAnalyzer adds an instance of the generated JPQueryTranslator1 subclass of JPQueryTranslator<Employee> as second parameter to the constructor of JPQuery<Employee>. The class name of the subclass generated by JPLambdaAnalyzer is derived as follows: the prefix is the class name of the class that contains the
JPQuery<T> instance followed by JPQueryTranslator and the line number in the source code with the associated JPQuery<T> instance. In the example above, we have omitted the class name before JPQueryTranslator1, because we have no class specified in Listing 6.10. This naming convention assists developers in locating a JPQuery<T> instance corresponding to a subclass of JPQueryTranslator<T>, and vice versa.
7 The JPGateway Component

The JPGateway component of CAPJa manages the connectivity of Java with a specific Prolog system. On the one hand, CAPJa provides a default gateway which is not specific to a given Prolog system. Therefore, it is called the Portable Prolog Gateway (PPG). On the other hand, it is possible to integrate into CAPJa already available Java-Prolog interfaces as custom gateways.

The remainder of this chapter reads as follows: Section 7.1 describes the abstract Java interface JPGateway which is exclusively referenced by the other components of CAPJa. Section 7.2 explains the dependencies between the JPGateway component and the remaining components of CAPJa for queries to Prolog. Section 7.3 presents the default gateway PPG of CAPJa and Section 7.4 discusses how available Java-Prolog interfaces can be integrated in CAPJa in the form of new gateways.

7.1 The JPGateway Interface

The JPGateway component is accessed by the other components of CAPJa exclusively via a single Java interface called JPGateway. The JPGateway interface abstracts the interaction of CAPJa with the JPGateway component and provides an uniform communication interface. The JPGateway interface specifies all necessary methods for the interaction with Prolog. For instance, it enforces the implementation of the methods that manage unification and backtracking in Prolog. We begin with a description of the abstract methods of JPGateway that are used for a direct interaction with Prolog. The method names were derived from the respective (ISO-)predicates with only a single argument in Prolog. An invocation of these methods leads to a direct call of the corresponding predicates in Prolog. The methods for the direct interaction with Prolog are as follows:

- boolean asserta(T t) can be invoked to assert a Java object in the Prolog database. In so doing the representation of t in Prolog is asserted as the first fact of the corresponding predicate in Prolog. The representation of t as term is derived by CAPJa’s default mapping mechanism.
• boolean `assertz(T t)` is used equivalently to the `asserta` method but asserts the representation of `t` in PROLOG as the last fact of the corresponding predicate in PROLOG.

• boolean `consult(Path path)` reads a file as PROLOG source file. The `Path` parameter is used to locate the file in a file system.

• boolean `retract(T t)` removes the first term that unifies with the representation of the parameter `t` in PROLOG from the PROLOG database.

• boolean `retractAll(Class clazz)` removes all terms that unify with the representation of the parameter `clazz` in PROLOG from the PROLOG database. A `Class` object is mapped similarly to PROLOG as any class instance. The only difference is that all mapped properties are represented by variables in PROLOG.

There are also additional version of the methods `asserta`, `assertz`, `retract`, and `retractAll` that have a second method parameter of type `JPMapper<T>`. The single-parameter versions use CAPJa’s default mapping mechanism to derive the representation of the passed method parameter `t`, the two-parameter versions use the passed mapper instance to derive the representations of `t` in PROLOG. To control the unification process in PROLOG, the `JPGateway` interface additionally enforces the implementation of the following five methods:

• `List<String>` `backtrack()` triggers backtracking for a called goal in PROLOG if PROLOG offers backtracking. This method returns a list consisting of variable bindings sorted by the occurrence of the variables in the called goal.

• `List<String>` `call(String goal)` calls a goal in PROLOG. This method returns a list consisting of variable bindings sorted by the occurrence of variables in `goal`.

• `List<List<String>>` `findall(String goal)` calls the `findall(+Template, :Goal, -Bag)` meta-predicate in PROLOG with `goal` as the second argument `Goal` of `findall`. A call to `findall` creates the PROLOG list `Bag` consisting of instantiations of `Template` which were obtained successively on backtracking over `Goal`. The return of the method is a list consisting of lists with variable bindings sorted by the occurrence of the variables in `goal`.

• boolean `isBacktrackable()` returns `true` if PROLOG offers backtracking, otherwise `false`.

• boolean `stopBacktracking()` stops backtracking if PROLOG offers backtracking.
Any gateway suitable for CAPJa has to implement the JPGateway interface and thus its abstract methods. The PPG is a default implementation of JPGateway which is represented by the PPGateway class in JAVA and will be clarified in Section 7.3. In the following section, we discuss the dependencies between the JPGateway component and the remaining components of CAPJa for queries to PROLOG.

7.2 Dependencies

All the components of CAPJa that are involved in the communication with PROLOG only rely on the JPGateway interface and thus are independent of an actual implementation. In this way, available JAVA-PROLOG interfaces can easily be integrated as custom gateways, without changing other components of CAPJa that manage the object-to-term mapping, queries, and the result handling. Figure 7.1 illustrates the JPGateway component and its dependencies within CAPJa. For readability, we have omitted class fields, parameterless constructors, private methods, and inherited methods. The affiliation of the pictured JAVA types to the components of CAPJa is indicated by surrounding frames. To be more specific, we have included all the necessary subtypes that are involved in the query from Listing 6.8 using the PPG. The respective JAVA types have bold lines in Figure 7.1.

A query from JAVA to PROLOG with CAPJa begins with an instance of JPQuery<T>, see also the exemplary query q1 of Listing 6.8 that is of type JPQuery<Employee> with Employee as query type. After an analysis of q1 by JPCompiler, the query q1 has a reference to JPQueryTranslator1 as additional, last constructor parameter. This subclass of JPQueryTranslator<Employee> implements the query constraints of q1 in the form of the three methods setVariableBindings, getSubgoals, and validate. Because we have not passed a JPMapper<T> instance as additional constructor parameter of q1, JPQueryTranslator1 applies CAPJa’s default mapping mechanism to the query type of q1. The responsible JPMapper<T> subclass is represented by the EmployeeJPDefaultMapper type in Figure 7.1. After an invocation of the setVariableBindings method of JPQueryTranslator1, the query type is translated by EmployeeJPDefaultMapper and the subgoals returned by the getSubgoals method of JPQueryTranslator1 are appended. A method of JPQuery<T>, e. g., getSolution, then executes the call via an instance of JPGateway that provides the connection with PROLOG. Each result returned by PROLOG is finally tested by the validate method of JPQueryTranslator1 in JAVA.

Figure 7.1 shows that all other components of CAPJa depend only on the abstract JPGateway interface and the JPGateway interface itself only depends on the abstract JPMapper<T> class. This clean separation ensures that the JAVA programs with
Figure 7.1: The JPGateway component and its dependencies within CAPJa.
CAPJa remain unaffected from an exchange of the used gateway and the connected prolog system.

### 7.3 The Portable Prolog Gateway

As already discussed in Chapter 3, former approaches to the integration of PROLOG and JAVA are usually specific to a particular PROLOG system. As a consequence, the used PROLOG systems cannot be exchanged easily, at least not without major adaption in JAVA. Because portability is an essential design goal of CAPJa, we have included the Portable Prolog Gateway (PPG) that is based on the JPGateway implementation PPGateway and provides connectivity with a host of different PROLOG systems. It has been already successfully tested with several PROLOG systems such as BPROLOG, the Ciao system, GNU-PROLOG, SWI-PROLOG, Tu-PROLOG, the XSB system, and YAPROLOG. Exchanging a PROLOG system connected via the PPG is an easy task and requires almost no modifications in JAVA.

The PPG is based on the streams standard input (stdin), standard output (stdout), and standard error (stderr). Standard streams are system input and output communication channels which enable a computer program to communicate with its environment. Originally, standard streams were designed to handle the input and output of physically connected devices, such as a keyboard for the input and a monitor for the output. However, nowadays, standard streams abstract this even more. If a command is executed via an interactive shell, the associated streams for input and output are typically connected to the text terminal on which the shell is running. This connection can be altered by redirections, e.g., via pipelines. The PPG uses these standard streams to interact with a PROLOG top-level. Figure 7.2 illustrates the communication of the PPG with PROLOG via the standard streams. The communication protocol in the process is plain PROLOG which requires no particular parsing on the PROLOG side except the usual interpretation, just as if a user has prompted a PROLOG goal to the PROLOG top-level. Therefore, the object-to-term mapping of CAPJa can be combined naturally with the PPG. Textual representations of terms that have been derived by the subclasses of JPMapper<T> and
JPQueryTranslator<T> already conform to Prolog’s syntax. With the help of the PPG, CAPJa is deployable for a broad range of operating systems and Prolog implementations. In addition, the PPG provides control over backtracking in Prolog.

Goals passed by an instance of JPQueryTranslator<T> via the PPG are piped to the standard input stream (stdin) of a connected Prolog process. Figure 7.3 illustrates the schematic flow of information between the individual pipelines. When opening a connection from a Java program to a Prolog top-level using the PPG, two pipelines are realized: outPipe for queries to Prolog and inPipe for variable bindings returned from Prolog. outPipe shares the same thread as the instance of JPQueryTranslator<T> and simply passes input to the stdin of the connected Prolog process. Queries from Java trigger the read-evaluate-print loop of the associated Prolog top-level. Variables bound by Prolog are returned via the used standard stream for output, i.e. stdout or occasionally stderr. inPipe runs in a separate thread which is represented by an instance of InputThread and reads the variable bindings from stdout (or stderr). In Java, variable bindings are passed as list to the actual instance of JPQueryTranslator<T> which converts them back to objects using the associated mapper.

**Implementation of the PPG.** The PPG is represented by the PPGateway class in Java. Because the PPG allows for an interaction with different Prolog systems, the user has to configure the PPG for the Prolog system that shall be used. For this purpose, the constructor of PPGateway offers a single parameter which expects an identifier in string format that is associated with the desired Prolog system. The passed reference refers to an entry of a configuration file in XML which
7.3 The Portable Prolog Gateway

has all installed PROLOG systems registered. Listing 7.1 shows an excerpt of the configuration file config.xml of the PPG and how an installed PROLOG system is represented in XML.

```
<prolog>
  <system name="XSB">
    <executable>C:/Program Files (x86)/XSB/bin/xsb64.bat</executable>
    <output>STDOUT</output>
  </system>
</prolog>
```

Listing 7.1: A single entry of the configuration file of the PPG.

The name attribute of the system element is used in the constructor of PPGateway to reference the corresponding PROLOG system. The child element executable locates the executable in the file system that is necessary to run the PROLOG system. In Listing 7.1, the executable for the XSB system is installed on a 64 bit Windows 7 platform. A referenced executable may also contain start-up parameters. The output element determines the stream corresponding to the output channel of the particular PROLOG system. Externalizing the configuration data to an XML-file simplifies and improves the accessibility of the PPG. Moreover, config.xml can also be utilized by CAPJa to store other user-defined configurations such as the PROLOG representation of null references in JAVA. In Section 5.1.3, we have defined the n_u_1_1 atom as default representation of null in PROLOG. However, externalizing the configuration of the PPG or CAPJa is completely optional. Instead of loading a configuration file and for reasons of performance, the necessary configuration data can also be integrated into CAPJa.

In order to work with CAPJa, the PPGateway implements the JPGateway interface and its methods. In the following, we describe selected methods of the PPGateway class. Listing 7.2 shows the implementation of the utility method connect which is used to start a PROLOG process but no method of the JPGateway interface.

```
public void connect() throws IOException, InterruptedException {
  disconnect();
  String exe = config.getExecutable();
  prologProcess = Runtime.getRuntime().exec(exe);
  outPipe = new BufferedWriter(
      new OutputStreamWriter(prologProcess.getOutputStream()));
  hasConnection = true;
}
```

Listing 7.2: The connect method of PPGateway.

First, any PROLOG process associated with the current PPG instance is disconnected. Then the location of the PROLOG executable in the file system is loaded.
The JPGateway Component

together with potential starting parameters. Thereafter, a runtime object is created that associates the current JAVA application and encapsulates the new PROLOG process. In Line 5 of Listing 7.2 the outPipe between the PPG and the PROLOG process is created. Any input to outPipe is written to the standard input of the PROLOG process. Finally, a flag for connectivity is set.

Listing 7.3 shows the implementation of the two-parameter version of the asserta method of PPGateway which calls the asserta/1 predicate in PROLOG as described in Section 7.1 and has a JPMapper<T> instance as second method parameter.

```
public synchronized <T> boolean asserta(T t, JPMapper<T> mapper) throws InterruptedException, IOException, PrologException {
    if (!isBacktrackable()) {
        inPipe = initInputThread(config.getOutput());
        write("asserta(" + mapper.getTermFromInstance(t) + ")");
        inPipe.join();
        boolean asserted = getStatus(result);
        result.clear();
        return asserted;
    }
    throw new IllegalStateException("Waiting for backtracking!");
}
```

Listing 7.3: Implementation of asserta of PPGateway.

Instead of CAPJa’s default mapping mechanism, the JPMapper<T> instance passed as second method parameter is used to map the instance t passed as the first method parameter. The method returns true if the instance t has been asserted successfully to PROLOG’s internal database. The invocation of this method may lead to an exception in JAVA, if another thread has interrupted the current thread or the I/O operations have failed. A PrologException can occur, if PROLOG returns an error. An error message from PROLOG is passed to the constructor of PrologException and then written to the JAVA console. The method is marked synchronized to prevent thread interference in multithreaded applications with CAPJa. In Line 3 of Listing 7.3, it is first tested if PROLOG is waiting for backtracking. If PROLOG is waiting for backtracking, an assert operation is not possible, which results in an IllegalStateException in JAVA. Otherwise, the inPipe is created receive messages from PROLOG. In Line 5 the getTermFromInstance method of the passed mapper instance is used to derive a textual representation of the method parameter t as term which then is written to PROLOG. The invocation of join forces the current thread to wait for inPipe to return (or die). In this way, we ensure that inPipe has completed the collection of incoming data from PROLOG. before we resolve the status in Line 7. The getStatus method processes the PROLOG result and returns the status. Finally, result is cleared.
Because the communication between the PPG and any PROLOG system is character-based, any top-level manipulation supported by the addressed PROLOG system can easily be implemented. The PPG currently only supports a minimal set of top-level manipulations which are commonly used by all tested PROLOG systems to control backtracking in PROLOG. If a query has an alternative solution, the PROLOG top-level prompts the user for input: start or stop backtracking. A single key stroke is used to start or stop backtracking, semicolon for the former and the ENTER key for the latter. Listing 7.4 shows the implementation of the backtrack method of PPGateway which is used to start backtracking in PROLOG.

```
public synchronized List<String> backtrack() throws InterruptedException, IOException, PrologException {
    if (isBacktrackable()) {
        inPipe = initInputThread(config.getOutput());
        write(";");
        inPipe.join();
        List<String> bindings = extractBindings(result);
        result.clear();
        return bindings;
    }
    throw new IllegalStateException("Backtracking not available!");
}
```

Listing 7.4: Implementation of backtrack of PPGateway.

The return of the backtrack method is a list of variable bindings from PROLOG. This method is implemented similarly to the asserta method, except for the following parts: If PROLOG is not ready for backtracking, an IllegalStateException is thrown. Otherwise, not a goal is submitted to PROLOG but a single semicolon in order to continue with backtracking. In Line 7, the variable bindings are extracted from the PROLOG return which is cleared afterwards.

**Performance of the PPG.** The price for the versatility of the PPG is an inferior performance compared to optimized JAVA-PROLOG interfaces that are specific to particular PROLOG systems. These interfaces usually have raw access to internals of the PROLOG systems or are completely integrated. However, the performance of the PPG is decent and compared to the only other portable approach LOGICOBJECTS [17], the PPG performs even better. Moreover, the performance gap between the PPG and other JAVA-PROLOG interfaces becomes marginal for more complex queries with a higher processing times in PROLOG.

We have used the PPG in a case study, see also Section 8.1, to execute a query system for the London underground on various PROLOG systems. In so doing the
used Java program with CAPJa has not been modified at all. The performance of the PPG in the case study will be discussed subsequently in Section 8.1.5. This discussion also includes a comparison of the PPG with LOGICOBJECTS and the high-performance JAVA-PROLOG interface JPL [89] for SWI-PROLOG.

7.4 Custom Gateways

There are many known models for the interaction of different programming languages. For instance, the PPG starts a PROLOG process from JAVA and communicates with a PROLOG top-level via standard streams for input and output. This approach is limited to operations on a single physical machine, which may not always meet the requirements. Therefore, CAPJa supports other communication approaches between PROLOG and JAVA simply by different implementations of the JPGateway interface in JAVA. In so doing programs based on CAPJa remain entirely unaffected. In the following, we will propose different approaches that are suitable for the use as custom gateways for CAPJa.

7.4.1 Remote Procedure Calls

In distributed computing, programs (clients) can remotely call procedures that are executed on another computer (server) in a shared network. This client-server interaction is based on request-response protocol which is embedded as a message-passing system. The functionality that clients can request from servers is referred to as service and ranges from simple data accessing to complex computations. The communication between client and server complies with a communication protocol, such as the well-known Hypertext Transfer Protocol (HTTP) or the Transmission Control Protocol (TCP). The client-server model can also be exploited for inter-language communication; e.g., for the integration of PROLOG and JAVA. The advantage of the client-server model is that it allows to run PROLOG and JAVA on different machines. Fortunately, CAPJa supports such an approach almost out of the box.

We will now present a client-server architecture, with JAVA as client and PROLOG as server, that can be integrated into CAPJa via a custom gateway that implements the communication protocol of the PROLOG server. Both programming languages, PROLOG and JAVA, provide program libraries to set up and request servers based on different communication protocols. In the following examples, we rely on the corresponding program libraries of SWI-PROLOG.
Prolog TCP Servers. We begin with a PROLOG server that provides a TCP socket as communication endpoint. To implement such a server in PROLOG, we use the library(socket) of SWI-PROLOG which facilitates TCP and UDP (User Datagram Protocol) inet-domain sockets. Listing 7.5 shows the `create_server(-Port)` predicate, which starts a server with a given Port in PROLOG. More information and examples on the usage of the socket library of SWI-PROLOG can be found in the official documentation [102].

```prolog
create_server(Port) :-
    tcp_socket(Socket),
    tcp_bind(Socket, Port),
    tcp_listen(Socket, 5),
    tcp_open_socket(Socket, AcceptFd, _),
    dispatch(AcceptFd).
```

Listing 7.5: Creation of a PROLOG server based on TCP sockets.

The Java package java.net provides the necessary classes and interfaces that can be used to create an infrastructure for networking in Java. For instance, the Socket class provides the methods to establish a socket-based communication in Java. Listing 7.6 shows a small example in Java where we open a socket and submit a request.

```java
try {
    Socket socket = openSocket("localhost", 8080);
    String result = request(socket, "plus(X,2,3), write(X)");
    System.out.println(result); // prints 1 to the Java console
    socket.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

Listing 7.6: Opening and requesting a socket from JAVA.

Suppose the PROLOG server is hosted locally with port 8080. First, we open a JAVA socket for the communication with the local PROLOG server. In Line 3 of Listing 7.6, we request the PROLOG server and call a simple goal in plain PROLOG. The resulting variable binding is written and redirected to the socket in Java. In contrast to the PPG, the writing of variable bindings to the socket in JAVA is necessary, because otherwise the variable bindings would not be available in JAVA. Finally, we print the result to the JAVA console and close the JAVA socket.

The implementation of a custom gateway to a TCP server running PROLOG requires just the following few steps. Any query created by CAPJa has to be extended by a subgoal with the write/1 predicate which instructs the PROLOG server to write all variable bindings contained in the actual query as a PROLOG list to the JAVA
The JPGateway Component

socket. Finally, individual variable bindings are extracted from the resulting String instance in Java. For this purpose, CAPJa already provides a Prolog parser based on the classes JPPrologTermParser and JPPrologTermListener, see also Section 5.3.2. The getMembers method of the JPPrologTermListener class extracts the members of the returned Prolog list in string format. Once the variable bindings are encapsulated by list in Java, we can process them in the same way as the PPG processes the return of a queried Prolog top-level.

Prolog HTTP Servers. Inter-language communication via a HTTP can be accomplished if we exploit the HTTP POST method which allows for arbitrary length instructions on a Prolog HTTP server. SWI-Prolog provides a series of program libraries for accessing HTTP servers as well as providing HTTP server capabilities [101]. The instructions contained in a POST message are handled on the server side like queries to a Prolog top-level. To start an HTTP server we use the http_server(:Goal, +Options) predicate of SWI-Prolog. The Options list must contain at least the term port(?Port) that determines the server’s port. Once the server is started, any client may send POST messages to the server by addressing the server’s URL and port. SWI-Prolog allows us to specify handlers for requests via the http_handler(+Path, :Closure, +Options) predicate where Closure references the predicate that implements the handler. Listing 7.7 shows the implementation of a handler that simply calls the term sent as POST message to the Prolog server.

```
:- http_handler('/call', handle_call, [spawn(http_pool), time_limit(infinite)]).

handle_call(Request):-
  member(method(post), Request),
  format('Content-type: text/xml; charset=utf-8~n~n'),
  http_read_data(Request, Data, []),
  term_to_atom(Term, Data),
  call(Term).
```

Listing 7.7: A simple handler implementation in SWI-Prolog for POST requests.

Listing 7.8 illustrates a POST message from Java to a local Prolog HTTP server where the handler from Listing 7.7 calls the plus/3 predicate in Prolog and writes the binding of the X variable to Java. The result is finally printed to the console in Java. In Java, we have used the DefaultHttpClient class of the Apache HttpClient open-source project [3] to request the Prolog server. This project is responsible for creating and maintaining a toolset of low level Java components focused on HTTP and associated protocols. The DefaultHttpClient class is a HTTP/1.1 compliant HTTP agent implementation and is configured for most common use cases.
7.4 Custom Gateways

Since the goal format of the POST message in Listing 7.8 is plain PROLOG, we can easily implement a custom gateway to HTTP servers in SWI-PROLOG. We have only to follow the same steps as explained for the custom gateway to TCP servers in SWI-PROLOG.

In contrast to the PPG, the approaches based on the TCP and HTTP libraries of SWI-PROLOG have a considerable drawback. The TCP and HTTP libraries do not support non-deterministic (remote procedure) calls out of the box. If a query has multiple solutions due to backtracking, then only the first solution is returned by the PROLOG server, all other solutions are discarded. However, using the findall/3 meta-predicate we can at least obtain all solutions at once. Due to the JPGateway interface, we have to implement the methods backtrack, isBacktrackable, and stopBacktracking. To reflect the mentioned limitation and to inform a user, the backtrack method could throw an UnsupportedOperationException in Java. The isBacktrackable method should always return false and the stopBacktracking method should return silently.

Pengines. In 2014, the Pengines (Prolog Engines) [47] package was introduced to SWI-PROLOG. It aims to simplify the development of JavaScript based web-applications that must talk to a PROLOG server and realize distributed programming in PROLOG by providing remote procedure calls (RPCs) over HTTP. Besides a high-level programming abstraction implemented on top of SWI-PROLOG’s thread predicates and HTTP client and server libraries, the package not only offers deterministic...
RPCs but non-deterministic RPCs as well. That is, we can solve queries and backtrack explicitly for alternative solutions. The communication protocol for Pengine is referred to as Prolog Transport Protocol (PLTP) which is based on communicating finite-state machines. The message transport format is plain PROLOG for a calling PROLOG process. For a calling JAVA SCRIPT process, the request is encoded in plain PROLOG and the response in JSON [44]. The minimalist way to create a custom gateway to a Pengine is to start a PROLOG process from JAVA and to address queries created by CAPJa to Pengine with the help of the pengine_rpc(+URL, +Query, +Options) predicate that executes Query on the Pengine referenced by URL. However, this requires a local installment of PROLOG. Another approach that requires no local installment of PROLOG is based on the HTTP server that backs a Pengine. We have implemented the JPPengine class which encapsulates a PROLOG Pengine in JAVA. A JPPengine instance is created by passing the URL and port information to locate the remote Pengine server. As part of the instance creation, a slave Pengine is initialized on the Pengine server and its id registered. As a gateway suitable for CAPJa, the JPPengine class has to implement the JPGateway interface. Relying on methods for HTTP POST and GET messages and a parser for JSON [1], we have partially reimplemented the PROLOG API of Pengine by the methods of the JPPengine class in JAVA. The method names in JAVA have been derived from the predicate names in PROLOG. Table 7.1 shows which predicates of the PROLOG API of Pengine have been used to implement the methods of the JPPengine class.

<table>
<thead>
<tr>
<th>JPPengine</th>
<th>Pengine Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>asserta</td>
<td>pengine_ask/3</td>
</tr>
<tr>
<td>assertz</td>
<td>pengine_ask/3</td>
</tr>
<tr>
<td>backtrack</td>
<td>pengine_next/2</td>
</tr>
<tr>
<td>call</td>
<td>pengine_ask/3</td>
</tr>
<tr>
<td>findall</td>
<td>pengine_ask/3</td>
</tr>
<tr>
<td>isBacktrackable</td>
<td>implicitly via pengine_ask/3 or pengine_next/2</td>
</tr>
<tr>
<td>retract</td>
<td>pengine_ask/3</td>
</tr>
<tr>
<td>retractAll</td>
<td>pengine_ask/3</td>
</tr>
<tr>
<td>stopBacktracking</td>
<td>pengine_stop/2</td>
</tr>
</tbody>
</table>

Table 7.1: Implementation of JPPengine based on PROLOG API of Pengine.

The careful reader probably misses the entry for the consult method. By default, predicates in PROLOG with unknown side-effects such as consult are considered unsafe by a Pengine. A Pengine is executed in the context of the sandbox module of SWI-PROLOG which provides the safe_goal(+Goal) predicate that tests whether a goal is safe to be called. However, there are ways to extend the set of safe predicates on a Pengine server, but not dynamically from JAVA.
7.4 Custom Gateways

7.4.2 CAPJa as a High-Level API

Most PROLOG systems offer raw access to foreign languages through integrated interfaces with high performance. Therefore, CAPJa can also be used as a high-level JAVA API for the PROLOG goal construction and the result handling together with a custom gateway that exploits an already established JAVA-PROLOG interface, such as JPL [89] for SWI-PROLOG or INTERPROLOG [15] for the XSB system.

CAPJa and JPL. JPL [89] uses the Java Native Interface (JNI) [62] and the Foreign Language Interface (Fli) [100] of SWI-PROLOG to establish a bidirectional communication channel between JAVA and SWI-PROLOG. The Fli is an interface between SWI-PROLOG and the C programming language. The Fli provides C functions to analyze, convert, and represent predicates in SWI-PROLOG. The Fli supports non-deterministic predicates and control over backtracking in SWI-PROLOG. Calls from C to PROLOG, and vice versa, can be arbitrary nested. JPL uses the Fli for communication between SWI-PROLOG and the JNI in JAVA. In addition to the laborious goal construction with the subclasses of Term in JAVA, JPL also accepts PROLOG goals in string format. Goals in string format are analyzed by JPL and converted internally. Listing 7.9 shows the different methods for calling a PROLOG goal in string format with JPL in JAVA.

```java
String goal = "member(X, [1,2,3])"
Query query = new Query(goal);
Map<String, Term> firstSolutionBindings = query.oneSolution();
Map<String, Term>[] allSolutionsBindings = query.allSolutions();
query.hasNext();
Map<String, Term> nextSolutionBindings = query.next();
q.close();
```

Listing 7.9: Java methods of JPL for calling PROLOG.

To obtain only the first solution we invoke the oneSolution method of the Query class. The same can be accomplished using the methods hasNext and next because the class Query implements the java.util.Iterator interface. The hasNext method returns true if JPL was able to call goal successfully within PROLOG. The next method allows us to iterate through all solutions of query within a loop over hasNext. To discard preemptively further solutions, we can simply close on open query. If we want PROLOG to return all possible solutions at once, we invoke the allSolutions method of the query class.

In order to implement the JPLGateway class that represents a custom gateway based on JPL and suitable for CAPJa, we have to implement the JPGateway interface. For this purpose, we have only to delegate method invocations of JPGateway to the
7 The JPGateway Component

corresponding methods of the Query class of Jpl. Table 7.2 clarifies which methods of the Query class of Jpl can be used to implement the necessary methods of the JPLGateway class.

<table>
<thead>
<tr>
<th>JPLGateway</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>asserts</td>
<td>oneSolution</td>
</tr>
<tr>
<td>assertz</td>
<td>oneSolution</td>
</tr>
<tr>
<td>backtrack</td>
<td>hasNext, next</td>
</tr>
<tr>
<td>call</td>
<td>hasNext, next</td>
</tr>
<tr>
<td>consult</td>
<td>oneSolution</td>
</tr>
<tr>
<td>findAll</td>
<td>allSolutions</td>
</tr>
<tr>
<td>isBacktrackable</td>
<td>hasNext</td>
</tr>
<tr>
<td>retract</td>
<td>oneSolution</td>
</tr>
<tr>
<td>retractAll</td>
<td>oneSolution</td>
</tr>
<tr>
<td>stopBacktracking</td>
<td>close</td>
</tr>
</tbody>
</table>

Table 7.2: Implementation of JPLGateway through Query of Jpl.

Because the results of queries with Jpl are of type Map<String, Term> or an array thereof, we have to extract the variable bindings and store them in a list of the type List<String> or List<List<String>> in order to comply with the return types of the methods of the JPGateway interface. Jpl does not support multiple PROLOG engines but CAPJa is intended to work with multiple PROLOG engines through multiple gateways; e.g., multiple instances of the PPG, see also Section 7.3. A PROLOG engine with default settings can be initialized with Jpl explicitly with the method init of the class JPL, which also provides methods to change the default settings. Because the Fl of Swi-PROLOG currently has only a no-op method to terminate a PROLOG session, it cannot be halted and reinitialized in a controlled manner. A PROLOG session started by Jpl dies automatically with the JAVA runtime. Therefore, we have to clarify this behavior in our gateway implementation for Jpl. Such a gateway can only be instantiated once. We achieved this, for instance, with the help of a static class property that is set upon the start of a new gateway. Unless the first gateway has been closed, any attempt to start a further gateway will throw an exception.

As shown later in Section 8.1.5, CAPJa achieves with JPLGateway a performance that is comparable to the performance of Jpl with Swi-PROLOG. At the same time, CAPJa relieves the programmer from the laborious goal construction and result conversion with Jpl in JAVA. Moreover, CAPJa’s automated object-to-term mapping mechanism and the Jpql enable clear, compact, and object-oriented queries to PROLOG with less programming effort as compared to Jpl, see also Section 8.1.5.
**CAPJa and Interprolog.** INTERPROLOG [15] is a bidirectional Java-Prolog interface for the XSB system. Additional support for Swi-Prolog has been claimed but apparently ceased after known issues with Version 5.4.6 of Swi-Prolog. INTERPROLOG exploits the Java’s Serialization API to map Java objects to terms in PROLOG, and vice versa. It can be used to create local or even remote PROLOG engines. A local PROLOG engine is accessed via the JNI, in a manner similar to JPL. The communication to a remote PROLOG engine is based on TCP/IP sockets.

Before we create a custom gateway based on INTERPROLOG and suitable for CAPJa, we first observe that PROLOG goals in INTERPROLOG can be passed in string format, see also Listing 7.10.

```java
PrologEngine engine = new XSBSubprocessEngine();
engine.command("import member/2 from basics");
Object[] bindings = engine.deterministicGoal(
    "findall(X, member(X,[1,2,3]), Bag), buildTermModel(Bag,BagM)",
    null, null, "[BagM]";
Object bindingOfBagM = bindings[0];
SolutionIterator it = engine.goal(
    "member(Y,[1,2,3]), buildTermModel(Y,YM)", "[YM]";
while (it.hasNext()) {
    Object bindingOfY = it.next()[0];
    it.cancel();
}
engine.shutdown();
```

Listing 7.10: Java methods of INTERPROLOG for calling PROLOG.

Therefore, queries with CAPJa can be passed to the methods of INTERPROLOG. For instance, the deterministicGoal method in Line 3 of Listing 7.10 tries to call the String instance passed as first method parameter as goal in PROLOG. The second parameter sets a PROLOG variable name that will be bound to the Object array passed as third parameter. With CAPJa, these method parameters are irrelevant and thus can be set both to null. In PROLOG, the buildTermModel(Term, TermModel) predicate of INTERPROLOG builds a TermModel object specification for any given PROLOG Term. We use this predicate to get a TermModel for the variable bindings of the actual query which we reference by the last method parameter of the deterministicGoal method.

The method deterministicGoal only considers the first solution. The goal method of INTERPROLOG calls a passed PROLOG goal in string format and immediately returns an instance of the SolutionIterator type which can then be used to obtain lazily the variable bindings of all solutions via the methods hasNext and next. If the call to PROLOG fails, the SolutionIterator instance has no elements. An invocation of the cancel method of SolutionIterator stops backtracking in PROLOG. As
with JPL, we can easily create the custom **IPGateway** for CAPJa that is based on INTERPROLOG. To implement the **IPGateway** interface we simply exploit the methods of the **PrologEngine** interface of INTERPROLOG. Table 7.3 shows which methods of **PrologEngine** are used to implement the necessary methods of the **IPGateway** class.

<table>
<thead>
<tr>
<th><strong>IPGateway</strong></th>
<th><strong>PrologEngine</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>asserta</td>
<td>deterministicGoal</td>
</tr>
<tr>
<td>assertz</td>
<td>deterministicGoal</td>
</tr>
<tr>
<td>backtrack</td>
<td>hasNext, next</td>
</tr>
<tr>
<td>call</td>
<td>hasNext, next</td>
</tr>
<tr>
<td>consult</td>
<td>deterministicGoal</td>
</tr>
<tr>
<td>findall</td>
<td>deterministicGoal</td>
</tr>
<tr>
<td>isBacktrackable</td>
<td>hasNext</td>
</tr>
<tr>
<td>retract</td>
<td>deterministicGoal</td>
</tr>
<tr>
<td>retractAll</td>
<td>deterministicGoal</td>
</tr>
<tr>
<td>stopBacktracking</td>
<td>cancel</td>
</tr>
</tbody>
</table>

Table 7.3: Implementation of **IPGateway** through **PrologEngine** of INTERPROLOG.

The integration of INTERPROLOG as a custom gateway allows CAPJa to call **JAVA** from **PROLOG** via the **PROLOG API** of INTERPROLOG but instead of **PROLOG** goals in string format with INTERPROLOG, CAPJa provides an object-oriented and powerful query mechanism to **PROLOG** in **JAVA**.
8 Evaluation

In this chapter, we evaluate our approach with CAPJa. Section 8.1 is a case study about a query system for the network of the London underground. We use CAPJa to integrate a PROLOG knowledge base for the network into JAVA. In this case study, we are interested, in how CAPJa’s integration and query mechanisms perform. For a better comparison, a reference implementation with JPL [89], we consider the well-known JAVA-PROLOG interface for Swi-PROLOG, too. For both approaches, we discuss the implementation effort and how it scales with the complexity of the queries. Furthermore, we are interested in how CAPJa’s abstraction layer affects the performance at runtime. Section 8.2 is a second case study which investigates how business rules can be integrated into JAVA. We implement the business rules in DATALOG* [87], an extension to standard DATALOG. From JAVA types, we derive a suitable fact base in DATALOG* which serves us as starting point for the development of the business rules. To bridge the gap between business analysts and programmers, the format of business rules is concise and declarative and we visualize their execution in DATALOG* by proof trees. Finally, Section 8.3 validates our approach with CAPJa from the applicability point of view. An in-depth comparison of these with the previously derived criteria for a seamless integration of PROLOG and JAVA from the object-oriented perspective and the discussed related work concludes this chapter.

8.1 Case Study on the London Underground

In this case study, we solve a problem suitable for logic programming: the implementation of a querying system for the network of a subway. For the user interaction we strive for a modern and intuitive graphical user interface for which JAVA is optimally suited. This is also a typical use case for the integration of a PROLOG application and a JAVA application.

In the following, we illustrate two different implementations of the case study. To compare them we count, for instance, the necessary physical source lines of code (LoC) [42]. LoC is a software metric which is used to measure a program’s size, it can also give rise to a program’s complexity. Smaller programs are often better readable and thus easier to maintain and refactor. We ignore LoC which only contain comments or a single opening or closing bracket. These LoC do not increase
8 Evaluation

a program’s complexity. The same is true for auto-generated LoC as they require no manual maintenance or revision.

8.1.1 The Prolog Knowledge Base

We have developed a knowledge base for the network of the London underground in Prolog. The network has been easily extracted with Prolog from data in GraphML, an XML-based file format for graphs. The graph for the London underground is directed. The stations of the underground are represented by the nodes of the graph. Stations are connected through lines which form the edges of the graph. In Prolog, we use the compact relation connected/3 to represent adjacent stations which are connected by an individual line. The connected facts suffice to represent the entire graph of the London underground. Listing 8.1 shows some exemplary facts.

| % connected(+Station1, +Station2, +Line) |
| connected(station('Bond Street'), station('Green Park'), line('Jubilee')). |
| connected(station('Green Park'), station('Westminster'), line('Jubilee')). |

Listing 8.1: Some Prolog facts about the London underground.

It is an easy task to describe the signature of the connected facts with the PSN in Prolog. Listing 8.2 shows the corresponding PSN expressions. We have used the functor/arity:className notation of the PSN to define the class name Connection which clearly is a more common class name in object-oriented programming, instead of Connected as class name. The same notation has been used to characterize the two different class members station1 and station2 which both associate the station/1 predicate in Prolog.

| % predicate(+Name, +Type, +Arguments) |
| predicate(connected/3:Connection, compound, [argument(station/1:station1, compound), argument(station/1:station2, compound), argument(line/1, compound)]). |
| predicate(station/1, compound, [argument(name, atom)]). |
| predicate(line/1, compound, [argument(name, atom)]). |

Listing 8.2: Signature of the fact base.

To enrich the knowledge base have added rules in Prolog. Listing 8.3 shows the compact implementation of the predicate


which proves if there is a tour from a starting station to a destination, using only the connections of an individual Line. Path is a Prolog list that contains all stations.
that have been visited during the tour. The list has Start as head and Destination as final member. If there is no tour between Start and Destination, the predicate fails. The reachable relation forms the transitive closure of the connected relation.

```
1 \% reachable(?Start, ?Destination, ?Line, ?Path)
2 reachable(Start, Destination, Line, Path) :-
3     connected(Start, S, Line),
4     reachable(S, Destination, Line, T),
5     Path = [Start|T].
6
7 reachable(Start, Destination, Line, [Start, Destination]) :-
8     connected(Start, Destination, Line).
```

Listing 8.3: The predicate reachable/4 in Prolog.

The signature of the predicate reachable/4 in Psn is shown in Listing 8.2. Because we already have defined the signatures of station/1 and line/1, we can omit their definitions here.

```
1 \% predicate(+Name, +Type, +Arguments)
2 predicate(reachable/4: 'Tour', compound,
3     [argument(station/1:start, compound),
4      argument(station/1:destination, compound),
5      argument(line/1, compound),
6      argument(path, list, station)]).
```

Listing 8.4: Signature of the reachable/3 predicate in Psn.

Given the Prolog fact base and the predicate reachable, we can already formulate several interesting queries about the London underground:

- q1: Is there a direct connection between two given stations? Through which line?
- q2: Is there a tour between a given starting station and a given destination?
- q3: How many tours are there with a given starting station and less than a given number of intermediate stops? What destinations do these tours have?

### 8.1.2 Queries from Java to Prolog

Former approaches which do not rely on a translation of Prolog programs to Java usually express queries from Java to Prolog in one of the following forms:

1. Pure Prolog code in Java string format.
2. A Java method which is tightly connected to a particular Prolog predicate.
3. An object-oriented term construction mechanism which allows us to represent the Prolog goals of a query in Java.
The common manner to express queries from Java to Prolog is in string format [15]. Except for escape characters, the query string in Java contains almost the same characters as the query in pure Prolog. However, a string-based query mechanism has considerable drawbacks. First of all, the approach is not object-oriented, and therefore, error-prone and difficult to debug in Java. Errors in the query string cannot be detected during compile time. Autocompletion or syntax highlighting is not available for strings and thus affects productivity. In addition, query parameters first have to be converted to strings which then have to be manually concatenated.

Queries that are based on particular Java methods such as the symbiotic methods in [17] are often less flexible. The usage of Java methods is limited compared to the usage of predicates in Prolog. Methods in Java always have fixed input parameters and only a single return type which contradicts the variability of predicates. Depending on the use case, predicate arguments may be used either for input or output. As a result, multiple, quite similar Java methods have to be implemented to cover each use case of a predicate. This may lead to naming issues because Java does not allow multiple methods with the same signature. Moreover, if multiple arguments of a predicate are intended for the output, a special method return type has to be introduced in Java in order to represent the Prolog output. Also simple subgoals based on predicate arguments require new methods in Java.

An object-oriented term-construction mechanism for goals in queries to Prolog is much more flexible. Particular Java types for Prolog, such as Atom, Variable, or Compound can be used to construct term in a bottom-up or top-down manner. For instance, to create the goal \( g(f(X)) \) bottom-up we first initialize the logical variable \( X \), then the compound \( f(X) \), and finally the compound \( g(f(X)) \). Top-down, works exactly opposite but appears to be less natural in Java. A notable issue of the term-construction mechanism is that it burdens the programmer with the translation of objects to terms, and vice versa. In addition, the necessary boilerplate code obfuscates the actual intention of querying Prolog and affects the readability considerable. Simple terms in Prolog already lead to complex and lengthy object-oriented representations in Java.

Because of the drawbacks of string format for Prolog queries in Java and the inflexibility of the coupling of Java methods and Prolog predicates, we have selected the term construction mechanism for an alternative implementation of the London underground example. In Section 8.1.4, we use the term construction mechanism of JPL [89], the custom Java-Prolog interface for Swi-Prolog, to query Prolog.
8.1.3 Implementation with CAPJa

To integrate the Prolog knowledge base for the London underground with CAPJa, only 7 LoC in Psn as shown in the Listing 8.2 and 8.4 have been necessary to generate the Java classes that represent the Prolog predicates in Java. In the same process, the corresponding mapper classes have been generated which implement the object-to-term mappings. The generated Java class that represents the reachable/4 predicate in Prolog can be found in Listing 8.5. Similar classes are generated by CAPJa for the predicates station/1, line/1, and connected/3.

```java
public class Tour {
    private Station destination;
    private Line line;
    private ArrayList<Station> path;
    private Station start;
    //...getter & setter
}
```

Listing 8.5: The Tour class in Java.

To formulate the queries q_1, q_2, and q_3 with CAPJa we use the JPQL. The implementations of all three queries are shown in Listing 8.6.

```java
JPQuery<Connection> q1 = new JPQuery<Connection>(
    connection -> connection.getStation1().getName() == "Bond Street"
    && connection.getStation2().getName() == "Green Park" );
Connection c = q1.getSolution();
if(c != null) {
    System.out.println(c.getLine().getName());
}
JPQuery<Tour> q2 = new JPQuery<Tour>(
    tour -> tour.getStart().getName() == "Bond Street"
    && tour.getDestination().getName() == "Westminster" );
Tour t = q2.getSolution();
if(t != null) {
    t.getPath().forEach(station -> System.out.println(station.getName()));
}
JPQuery<Tour> q3 = new JPQuery<Tour>(
    tour -> tour.getStart().getName() == "Bond Street"
    && tour.getPath().size() < 4 );
List<Tour> tours = q3.getAllSolutions();
if(tours != null) {
    tours.forEach(tour -> System.out.println(tour.getDestination().getName()));
}
```

Listing 8.6: Queries to the London Underground with CAPJa in Java.
With CAPJa, the solutions of queries to PROLOG are encapsulated as instances of the query type. The implementations of $q_1$, $q_2$, and $q_3$ in Listing 8.6 contain also a print command for the solutions. All three queries together only require 16 LoC in Java. If an integrated development environment with autocomplete functionality is used, the implementation of the queries with CAPJa is a short affair. Moreover, because the queries are purely object-oriented with CAPJa, they require no proficiency with PROLOG.

### 8.1.4 Implementation with JPL

Basically, there is no integration step for PROLOG predicates in Java with JPL. The Jpl approach assumes that the Java programmer implicitly knows the signature of all involved predicates in PROLOG. With Jpl, terms are created bottom-up in Java. The query results with Jpl are returned as `Map<String,Term>` instance where each variable of the query is mapped to its binding. However, if we want to deal with the same meaningful Java types than before (`Station`, `Line`, and `Tour`), we first have to implement them. In addition, the conversion of variable bindings is also left to the programmer.

Fortunately, we can reuse the classes that we already have generated with CAPJa. For the use with JPL, we still have to add converter facilities between objects and terms. In order to keep the queries with JPL as compact as possible, we have decided to implement the converter facilities as methods of the relevant classes. In addition, we add parameterized constructors for a compact creation of objects. Listing 8.7 shows the two converter methods for the `Station` class. Similar methods are necessary for the `Line` class.

```
public static Term toTerm(Station s){
    return new Compound("station", new Term[]{new Atom(s.getName())} );
}

public static Station fromTerm(Term t){
    return new Station(t.arg(1).name());
}
```

Listing 8.7: Translation methods for the `Station` class with JPL.

Now, we can implement the three queries from above with JPL. Listing 8.8 shows the implementation of the query $q_1$. Instead of initializing the `Station` objects, we could have created the corresponding object-oriented representations for terms with JPL and used them directly for the queries to PROLOG. However, then we would have to translate the terms back to `Station` objects.
8.1 Case Study on the London Underground

Station station1 = new Station("Bond Street");
Station station2 = new Station("Green Park");
Term goal1 = new Compound("connected", new Term[]{Station.toTerm(station1),
    Station.toTerm(station2), new Variable("Line") });
Query q1 = new Query(goal1);
Map<String, Term> bindings1 = q1.oneSolution();
if(bindings1 != null) {
    Line line1 = Line.fromTerm(bindings1.get("Line"));
    Connection c = new Connection(station1, station2, line1);
    System.out.println(c.getLine().getName());
}

Listing 8.8: Query $q_1$ with JPL in JAVA.

The following listing shows the implementation of the query $q_2$.

Station station1 = new Station("Bond Street");
Station station3 = new Station("Westminster");
Term goal2 = new Compound("reachable", new Term[]{Station.toTerm(station1),
    Station.toTerm(station3), new Variable("Line"), new Variable("Path") });
Query q2 = new Query(goal2);
Map<String, Term> bindings2 = q2.oneSolution();
if(bindings2 != null) {
    Line line2 = Line.fromTerm(bindings2.get("Line"));
    Term[] pathTerm = Util.listToTermArray(bindings2.get("Path"));
    ArrayList<Station> path = new ArrayList<>();
    for(Term stationTerm : pathTerm) {
        path.add(Station.fromTerm(stationTerm));
    }
    Tour t = new Tour(station1, station3, line2, path);
    t.getPath().forEach(station -> System.out.println(station.getName()));
}

Listing 8.9: Query $q_2$ with JPL in JAVA.

The implementation of query $q_1$ with JPL requires 10 LoC. With CAPJa, we only need 6 LoC. With the increasing complexity of the queries, the gap in terms of LoC between JPL and CAPJa increases, too. Query $q_2$ requires 14 LoC with JPL and only 6 LoC with CAPJa.

Listing 8.10 finally shows the implementation of the query $q_3$ which has the highest complexity of all three queries. We require 23 LoC with JPL, with CAPJa again only 6 LoC are sufficient for the implementation.

Station station1 = new Station("Bond Street");
Variable path3 = new Variable("Path");
Term subgoal1 = new Compound("reachable", new Term[] { Station.toTerm(station1),
    new Variable("Destination"), new Variable("Line"), path3 });

149
8 Evaluation

```java
Variable length = new Variable("Length");
Term subgoal2 = new Compound("length", new Term[] { path3, length });
Term subgoal3 = new Compound("<", new Term[] { length, new org.jpl7.Integer(4) });
Term goal3 = new Compound("", new Term[] { subgoal1, new Compound("", new Term[] { subgoal2, subgoal3 }) });
Query q3 = new Query(goal3);
Map<String, Term>[] solutions = q3.allSolutions();
if (solutions != null) {
    ArrayList<Tour> tours = new ArrayList<Tour>();
    for (Map<String, Term> bindings3 : solutions) {
        Term[] pathTerm = Util.listToTermArray(bindings3.get("Path"));
        ArrayList<Station> path = new ArrayList<>();
        for (Term stationTerm : pathTerm) {
            path.add(Station.fromTerm(stationTerm));
        }
        tours.add(new Tour(station1, Station.fromTerm(bindings3.get("Destination")), Line.fromTerm(bindings3.get("Line")), path));
    }
    tours.forEach(tour ->
        System.out.println(tour.getDestination().getName()));
}
```

Listing 8.10: Query q3 with JPL in Java.

In the next section, we will compare the approaches with CAPJa and JPL.

### 8.1.5 Results

This section discusses the results of the London underground case study. A similar case study with LOGICOBJECTS and JPL can be found in [17], we therefore have included these results in our discussion.

**Programming effort.** The following table summarizes the necessary LOC for the integration of the PROLOG knowledge base and the implementation of the queries with CAPJa and JPL. When not stated otherwise, the numbers in Table 8.1 are LOC in JAVA.

<table>
<thead>
<tr>
<th></th>
<th>Integration</th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPJa</td>
<td>7 (in PROLOG)</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>JPL</td>
<td>48</td>
<td>10</td>
<td>14</td>
<td>23</td>
<td>69</td>
</tr>
</tbody>
</table>

Table 8.1: Necessary LOC with CAPJa and JPL.
As compared to the approach with JPL, Table 8.1 clearly shows that the approach with CAPJa leads to a significant reduction in LoC. The implementations of the queries with JPL require between 66% and 283% more LoC than with CAPJa. Therefore, the programming effort with JPL is significantly higher than with CAPJa. Moreover, the programming effort with JPL scales poorly with an increasing complexity of the queries. With CAPJa, we always need 6 LoC for each query. Because there is no predicate integration mechanism available for JPL, the necessary Java classes such as Station, Line, and Tour have to be implemented by the programmer. In addition, these classes require additional parameterized constructors for compact object creations and additional converter methods for a convenient translation between objects and terms.

However, it should be mentioned that the usage of LoC for the measurement of the complexity of a program is controversially discussed. Therefore, we strongly emphasize that the object-oriented interface to the Prolog knowledge base that has been generated with the help of CAPJa, consists of very simple Java classes. The queries in JPQL are only based on these generated Java classes and, therefore, the queries in JPQL are very clear and concise. In this way, CAPJa offers in Java an user-friendly interface to Prolog. If the concepts of the classes Tour, Station, and Line are clear, the queries can be formulated without proficiency in Prolog. Reasoning in Prolog remains completely hidden. On the other hand, with the PSN Prolog developers can conveniently characterize in Prolog how a Prolog knowledge base can be accessed from Java. This supports collaborative software development without paradigmatic interferences.

The similar case study in [17] with LOGICOBJECTS and JPL shows that the approach with LOGICOBJECTS also leads to a reduction in LoC if compared to JPL. However, the evaluation of the case study with LOGICOBJECTS has only considered the LoC in Java, the necessary LoC in Logtalk which add a significant layer to Prolog have not been included. With CAPJa, there is no adaption in Prolog necessary.

**Performance.** The fact base for the complete London Underground consists of 412 connections with 297 stations and 13 lines. To compare the performance of the implementations with CAPJa and JPL, we have measured\(^\text{10}\) the total execution time for 50000 consecutive calls of each query. We have used the PPG to establish a connection between Java and Swi-Prolog. The results of the performance tests can be found in Table 8.2. The execution times are in seconds.

In comparison to JPL, the execution with the PPG is slower. The simplest query $q_1$ takes with the PPG approximately 5.95 times longer than with JPL. However,\(^\text{10}\) Core i7-3720QM CPU @2.60GHz, 8GB RAM, Windows 7 - 64 bit
with only 0.25 milliseconds on average this still is very fast. The most complex query $q_3$ with the PPG only takes 1.8 times longer than with JPL. This shows that the PPG has a decent performance with SWI-PROLOG. For more complex queries, the performance differences between the PPG and JPL are almost mitigated.

Similar performance tests in [17] with LOGICOBJECTS and JPL have shown that LOGICOBJECTS has been much slower than JPL. The LOGICOBJECTS approach has been up to 26 times slower than JPL. Of particular importance is that LOGICOBJECTS has internally used JPL to interface the LOGTALK extension for SWI-PROLOG. Therefore, we have included the execution times of the JPLGATEWAY in Table 8.2. JPLGATEWAY is a custom gateway for CAPJa that exploits JPL to establish a connection between JAVA and SWI-PROLOG, see also Section 7.4.2. The execution times with JPLGATEWAY are only 10% to 30% slower than with JPL. This is significantly faster than with LOGICOBJECTS and proves also how little CAPJa’s abstraction layer in JAVA affects the overall runtime performance. Moreover, the JPLGATEWAY approach nicely shows that CAPJa can simply combined with already available JAVA-PROLOG interfaces for an improved performance.

Because the PPG allows for connectivity with different PROLOG systems, we have extended our tests to include the following freely available PROLOG systems: BPROLOG, the CIAO system, GNU-PROLOG, TU-PROLOG, the XSB system, and YAProLOG. For this purpose, no modifications have been necessary to our JAVA program with CAPJa. We have tested again 50000 consecutive calls of the queries $q_1$, $q_2$, and $q_3$. The execution times of the different PROLOG systems together with the PPG are presented in Table 8.3. All execution times are given in seconds. For a better comparison, we have included, again, the results of the PPG with SWI-prolog.

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>CIAO</th>
<th>GNU</th>
<th>SWI</th>
<th>TU</th>
<th>XSB</th>
<th>YAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>7.9</td>
<td>24.7</td>
<td>8.1</td>
<td>12.5</td>
<td>9.0</td>
<td>17.5</td>
<td>6.9</td>
</tr>
<tr>
<td>$q_2$</td>
<td>141.6</td>
<td>137.1</td>
<td>65.4</td>
<td>61.0</td>
<td>167.6</td>
<td>64.1</td>
<td>11.1</td>
</tr>
<tr>
<td>$q_3$</td>
<td>161.5</td>
<td>161.8</td>
<td>76.3</td>
<td>87.5</td>
<td>275.7</td>
<td>99.5</td>
<td>21.2</td>
</tr>
</tbody>
</table>

Table 8.3: Performance of the PPG with various PROLOG systems.
The slowest test has been with Tu-Prolog for $q_3$. It took 275.7 seconds for 50000 executions which is on average 5.51 milliseconds for a single execution. This is still decent and more than sufficient for most multi-paradigm programs with with Prolog and Java. Moreover, the main objective of the PPG has been portability, in its current status there is still potential for further improvements which will also increase the performance of the PPG.

8.2 Case Study on Taxes in International E-Commerce

Electronic commerce (e-commerce) has experienced an accelerated growth in the last years. In order to keep the prices of traded goods as low as possible and to provide shopping services around the clock via the Internet, internal business processes such as the recording of orders in financial accounting have to be automated. Rule-based systems can support the automation of business processes, for instance by automated decision making in financial accounting.

8.2.1 Logic Programming for Business Rules

Detailed knowledge about internal business processes is essential for any company and even more so if they have their business processes automated by a software system. Most software systems have been developed over a long period of time, and the exact parts of a program that reflect business processes are often hard to extract. Therefore, refactoring and updating the internal business processes, or even understanding how they are modeled, may prove to be a considerable challenge. Another problem is the gap between business analysts and developers that have to implement the requirements made by the business analysts. To bridge this gap and to make complex business logic more transparent, business rules have become more and more popular for business process management [19].

Business rules are statements that define or restrict certain aspects of a business process [14]. While business rules can occur purely informally, a careful, unambiguous, and consistent formulation of the rules is particularly important. Business rules can help to avoid cost-intensive misconceptions, to improve the communication, to ensure legal regulations, and to improve customer satisfaction [97]. Moreover, business rules can be used to separate business logic clearly from the rest of a business application. However, due to an increasing complexity in software development communication problems occur often between software developers and business analysts. Both need a common language for the problem domain. In fact, business analysts should have the possibility to formulate process logic specifications on a formal, abstract level.
In practice, it is a long, iterative process to model the application domain in cooperation with the customer (domain expert), and to develop the business rules. There are many communication problems and misunderstandings. It is crucial that the business rules are formal and to some extent understandable for the customer as well. Business rules in connection with domain-specific languages (DSL) probably can bridge the gap between customers, business analysts, and developers.

Logic programming languages such as PROLOG allow us to formulate declarative business rules in a compact format. Moreover, PROLOG is well suited to develop entire domain-specific languages for business analysts and software developers alike. Domain specific languages are formal languages that usually cover only a limited area (of the business logic) but may serve as an unambiguous medium for communication. DSLs have already proven reasonable to simplify and clarify several areas of software development [31]. We can use techniques from logic programming which have been used for a long time to support abstraction such as meta-programming in PROLOG. Therefore, building a domain-specific language with logic programming can be a much simpler task than the standard approach where compilers need to be developed. Fortunately, the syntax and semantics of PROLOG allow for particularly readable domain-specific languages.

Usually, business rules have to be evaluated in a bottom-up, forward chaining manner. For this purpose, we use the DDK package DATALOG* [87]. Compared to standard DATALOG, the extension DATALOG* allows for a larger set of connectives (including conjunction and disjunction), function symbols, and stratified PROLOG predicates (including aggregation and default negation) in rule bodies. DATALOG* programs are evaluated bottom-up, just like standard in DATALOG. With the help of CAPJa, also DATALOG* programs can be integrated into JAVA. During the development, a visualization of the program execution in DATALOG* can be very helpful. For this purpose, DATALOG* has been extended [84, 85] to generate proof trees [18]. Proof trees can be used to visualize the derivation process and are obtained based on (automatic) program transformation. Moreover, the abstraction of the program execution as proof tree further improves the understanding between business analyst and developer.

In the following, we illustrate how CAPJa simplifies the integration of business rules in DATALOG* into JAVA. The described use case is based on our former work on logic programming for building, updating and testing complex business logic [66]. We use business rules in DATALOG* to improve an e-commerce system. The business logic of the original system was implemented overly complicated in JAVA and difficult to extract. With the help of logic programming, we have reimplemented and extended
the business logic. The business rules are used to automate the handling with the delivery threshold for international trading in the European Union.

### 8.2.2 Delivery Threshold

In the European Union, the sales tax for shipments to foreign countries usually has to be paid in the home country. However, the sales tax has to be paid to the country of dispatch as soon as the accumulated shipments abroad exceed a certain net merchandise value per year. We call this limit the *delivery threshold*. For instance, if the annual net merchandise value of shipments from Germany to France exceeded 100,000 euros (until the end of 2015), the sales tax had to be paid in France. In 2016, France lowered the delivery threshold to 35,000 euros. From the legal texts [13] on the delivery threshold the following two rules can be extracted:

1. If the delivery threshold has exceeded last year or before this date, then the sales tax has to be paid in the country of dispatch. For this year’s calculations, a preliminary profit of a currently processed invoice has to be included. To determine the preliminary profit, the sales tax of the home country has to be accounted for.

2. If the first rule does not apply, the sales tax has to be paid in the home country.

It is a difficult task to write clear and concise rules. The underlying law text has four paragraphs with redundant conditions. Formulating rules without ambiguity and redundancy is essential for a refined representation of knowledge. For this purpose, logic programming offers the perfect tools.

In this case study, the task of the business rules is to calculate the resulting bookings for an already paid invoice. We are interested in the sales amount, the type of taxation, and the country of dispatch. At any one time, only a single given invoice is processed by the business rules.

### 8.2.3 Business Objects

The relevant business objects are invoices, annual sales, bookings, taxes, countries, and delivery thresholds. Invoices, annual sales, and bookings are represented by classes in JAVA. Taxes, countries, and delivery thresholds are defined as predicates in DATALOG.

In the original e-commerce system, an invoice is already represented by classes in JAVA. Because the classes associated with an invoice are also used by other modules of the application, they cannot be modified for the use with the business rules. The
UML class diagram in Figure 8.1 illustrates the necessary JAVA types to represent an invoice in JAVA. We have omitted constructors and methods in the class diagram to keep them more clear and because they are not relevant to our problem. Invoices serve as input for the reasoning process in DATALOG*. Bookings resulting from a processed invoice are returned as instances of the Booking class, see also Figure 8.2. In [66], the results of a query to the business rules were returned as XML document. The elements of the XML document had a similar structure as the Booking class and were subsequently converted to Booking instances in JAVA.

With CAPJa, we need no intermediate communication layer based on XML, we can customize the representations of relevant class instances as terms that are suitable for use with DATALOG*. Theses terms, then, can be used as input for the business rules in DATALOG*. Because we already have suitable JAVA types, we approach the integration problem from the JAVA perspective. In JAVA, CAPJa provides two options to define custom representations of class instances as terms in PROLOG. We can use CAPJa’s @JPMapping annotation layer or alternatively the JpML. For companies, it is often more conceivable that new components come with no modifications to existing source code. Therefore, we have decided to use the JpML.

Listing 8.11 shows the custom mapping definitions with JpML encapsulated by methods for all four relevant JAVA types. Because the class names can be translated by CAPJa canonically to functors, the mapping definitions require no further specifications of the functors, i.e. an instance of the Invoice class is translated to a term with invoice as functor. The JPMapper<T> instances that are returned by the
methods of Listing 8.11 translate the instances of the associated mapping type to corresponding terms in DATALOG∗, and vice versa. The generated custom mappers all have the corresponding mapping type as prefix and JPCustomMapper as suffix.

```java
public JPMapper<AnnualSales> getAnnualSalesMapper() {
    return new JPMapperFactory<AnnualSales>().create("1",
            (a, t) -> {t.setArgumentOrder(a.getDestination(), a.getYear(),
                a.getTotal());
            });
}

public JPMapper<Booking> getBookingMapper() {
    return new JPMapperFactory<Booking>().create("1",
            (b, t) -> {t.setArgumentOrder(b.getLine(), b.getType(), b.getMode(),
                b.getCountry(), b.getAmount());
            });
}

public JPMapper<Invoice> getInvoiceMapper() {
    return new JPMapperFactory<Invoice>().create("1",
            (i, t) -> {t.setArgumentOrder(i.getDestination(), i.getYear());
            });
}

public JPMapper<Position> getPositionMapper() {
    return new JPMapperFactory<Position>().create("1",
            (p, t) -> {t.setArgumentOrder(p.getLine(), p.getType(), p.getTotal());
            });
}
```

Listing 8.11: Custom mapping definitions with JPML.

The business objects in DATALOG∗ are represented by two groups of predicates. The first group is provided by JAVA and has been derived from invoices and the annual sales for last and this year. Listing 8.12 shows some facts which result from instances of the classes AnnualSales and Invoice.

```java
% annual_sales(+Destination, +Year, +Total)
annual_sales(’France’, 2014, 92300).
annual_sales(’France’, 2015, 99800).
% invoice(+Destination, +Year)
invoice(’France’, 2015).
% position(+Line, +Type, +Total).
position(1, food, 211.00).
position(2, nonfood, 120.00).
```

Listing 8.12: Facts provided by JAVA.
The facts for annual_sales/3 provide the business volume for a single country in a given year. An invoice/2 fact and the associated position/3 facts are derived from a single instance of the Invoice class in Java.

The second group is part of the configuration in Datalog*. The facts for tax/3 provide the information about the taxes for a given country, i.e., the tax rates for the different tax types. We only consider the two types food and nonfood according to the simplified enumeration TaxType in Java. The facts for delivery_threshold/3 determine a country’s annual delivery threshold. For our current invoice, only the country France is of interest and the home country Germany of the trader.

```
% tax(+Country, +Type, +Rate)
tax('France', food, 0.055).
tax('France', nonfood, 0.200).
tax('Germany', food, 0.070).
tax('Germany', nonfood, 0.190).
% delivery_threshold(+Destination, +Year, +Threshold)
delivery_threshold('France', 2015, 100000).
% home_country(+Country)
home_country('Germany')
```

Listing 8.13: Facts for the configuration in Datalog*.

### 8.2.4 Business Rules

The rule updated_annual_sales/3 in Listing 8.14 aggregates the business volume for the current year including the given invoice.

```
% updated_annual_sales(+Destination, +Year, -Total)
updated_annual_sales(Destination, Year, Total) :-
    dbbase_aggregate([Destination, Year, sum(Value)],
        ( annual_sales_so_far(Destination, Year, Value)
            ; position_net_value(Destination, Year, Value) ),
        Tuples ),
    member([Destination, Year, Total], Tuples).
% position_net_value(+Destination, +Year, -Net_Value)
position_net_value(Destination, Year, Net_Value) :-
    invoice(Destination, Year),
    position(_, Type, Total),
    home_country(Home),
tax(Home, Type, Rate),
Net_Value is Total/(1+Rate).
```

Listing 8.14: Aggregation of the current business volume.
We use the `ddbase_aggregate/3` predicate of the DisLog Developers’ Kit (DDK) [82] for the aggregation in `updated_annual_sales/3`. The business volume given by `annual_sales/3` is added to the net value of the invoice, which is derived by `position_net_value/3`. To obtain the net value of a position in an invoice, we use `tax/3` for the home country. The `tax_country/1` predicate in Listing 8.15 implements the two business rules for the delivery threshold and determines the country where the taxes have to be paid.

```
% tax_country(-Country)
tax_country(Country) :-
    invoice(Destination, Year),
    ( ( Y is Year - 1 ; Y is Year ),
        updated_annual_sales(Destination, Y, Total),
        delivery_threshold(Destination, Y, Threshold),
        Total > Threshold ->
        Country = Destination
    ; home_country(Home),
    Country = Home).
```

Listing 8.15: Business rules for the delivery threshold.

With the information of the derived taxing country, the rule for `booking/5` calculates the amount for every position of an invoice, separated by the two modes `profit` and `taxes`.

```
% booking(-Line, -Type, -Mode, -Country, -Amount)
booking(Line, Type, Mode, Country, Amount) :-
    invoice(Destination, _),
    position(Line, Type, Total),
    tax_country(Tax_Country),
    tax(Tax_Country, Type, Rate),
    ( Mode = profit, Country = Destination,
        Amount is Total/(1+Rate)
    ; Mode = tax, Country = Tax_Country,
        Amount is Total*Rate/(1+Rate) ).
```

Listing 8.16: Bookings corresponding to a position of an invoice.

Upon request from JAVA, derived bindings of `booking/5` are returned and converted to instances of the `Booking` type in JAVA. The business rules, together with the predicates for the configuration, are saved in the text file `bookings.pl`. The text file can be loaded by DATALOG∗ which automatically asserts all predicates and facts included in the text file to the internal database of DATALOG∗.
8.2.5 Calling the Business Rules

The business rules are evaluated in a bottom-up manner in DATALOG*. The predicate position_net_value/3 has to be evaluated before updated_annual_sales/3; this precedence is due to the meta-predicate ddbase_aggregate/3 for aggregation. Another well-known meta-predicate that requires stratification in deductive databases is default negation not/1. In principle, the other rules could be evaluated simultaneously bottom-up. In our system, however, they are evaluated subsequently. Most of the used predicates are deterministic for a given input. The all-results-inference-capability is especially essential for the two non-deterministic predicates position_net_value/3 and booking/5.

Listing 8.17 shows the method getBookingsFromInvoice in Java. An invocation retrieves all bookings according to the positions of a given invoice. Input parameters are a given invoice and the business volumes for the last and the actual year. The return is a list of Booking instances which have been derived by DATALOG*. We use the default interface PPG of CAPJa to establish a connection with DATALOG*. Therefore, we have added the appropriate entries to the config.xml used by the PPG that cover the initialization of a DATALOG* process. In Line 4, the business rules and the configuration are loaded. From Line 5 to 11 the input parameter of the methods are asserted to DATALOG*. Finally, we query DATALOG* for all resulting bookings and end the DATALOG* process.

```java
public List<Booking> getBookingsFromInvoice(Invoice invoice, AnnualSales lastYear,
                                           AnnualSales thisYear) {
    PPGateway datalogs = new PPGateway("DATALOGS");
    datalogs.connect();
    datalogs.consult("bookings.pl");
    JPMapper<AnnualSales> asm = getAnnualSalesMapper();
    datalogs.assertz(asm, lastYear);
    datalogs.assertz(asm, thisYear);
    JPMapper<Position> pm = getPositionMapper();
    invoice.getPositions().forEach(position -> datalogs.assertz(pm, position));
    ArrayList<Booking> bookings = new JPQuery<Booking>(getBookingMapper())
        .getAllSolutions(datalogs);
    datalogs.disconnect();
    return bookings;
}
```

Listing 8.17: Accessing the business rules in DATALOG* from Java.

**Example of Calculation.** In the following, we process the invoice given in Listing 8.12 with the configuration from Listing 8.13 step by step. First, the updated
business volume is determined with the help of updated_annual_sales/3. We summarize the derived facts below.

1. updated_annual_sales('France', 2014, 92300).
2. updated_annual_sales('France', 2015, 100098).

As one can see, the delivery threshold for 2014 was not exceeded, but for 2015 it was. Apparently, the company’s profit has increased between 2014 and 2015. The annual sales to the destination country France, including the currently processed invoice, exceed the delivery threshold in 2015. Therefore, the tax has to be paid in France.

1. tax_country('France').

Finally, booking_position/5 calculates the resulting booking positions for the modes profit and tax:

1. booking(1, food, profit, 'France', 200.00).
2. booking(1, food, taxes, 'France', 11.00).
3. booking(2, nonfood, profit, 'France', 100.00).
4. booking(2, nonfood, taxes, 'France', 20.00).

The derivation of the first fact for booking_position/5 is visualized as a proof tree, see Figure 8.3. The rule nodes in the proof tree are labeled by numbers, and they are shown as blue boxes. The nodes for the derived atoms are shown as red circles, that are labeled with the atoms. Basic predicates from the configuration or calls to built-in predicates (such as is/2 and >/2) are given by nodes in the form of an orange circle.

The visualization of the proof trees can be configured by the developer. Firstly, it is possible to group body atoms in the proof trees for better readability; these atoms are then joined by an and node, that is depicted as a white rhombus. Secondly, trivial body atoms can be excluded to reduce the complexity of the visualization; e. g., the atoms for invoice/2 and delivery_threshold/3 are excluded since they are part of the input and are not modified during the computation. Thirdly, the included atoms can be abbreviated suitably. For a better overview, we use abbreviated predicate names in the proof trees; e. g., the predicate symbol updated_annual_sales/3 is abbreviated to sales/3, tax_country/1 is abbreviated to country/1.

A proof tree is encoded in a term structure that resembles XML. This term structure is processed with the query, transformation, and update language FnQUERY [83] which is also part of the DDK. The term structures for proof trees do not lead to any termination problems even for recursive rule sets, since it is not allowed that
8 Evaluation

Figure 8.3: Visualization of the program execution with proof trees.

an atom is used in its own derivation during the bottom-up evaluation. This can also be tested by investigating the proof tree.

If we assume for 2015 a lower value for annual_sales/3, e. g., 80,000 euros, then the updated annual sales to the destination France no longer exceeds the delivery threshold in 2015 and the tax has to be paid in the home country Germany. The derived bookings, then, are as follows:

<table>
<thead>
<tr>
<th>Booking Number</th>
<th>Type</th>
<th>Description</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>booking</td>
<td>(1, food, profit, 'France', 197.20)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>booking</td>
<td>(1, food, taxes, 'Germany', 13.80)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>booking</td>
<td>(2, nonfood, profit, 'France', 100.50)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>booking</td>
<td>(2, nonfood, taxes, 'Germany', 19.10)</td>
<td></td>
</tr>
</tbody>
</table>

8.2.6 Results

This case study demonstrates another cooperation scenario of object-oriented and logic programming and proves that logic programming can be exploited for the development of business rule systems. Moreover, logic programming offers an alternative, declarative approach to business rules and has tools to improve the communication between business analysts and developers, for instance the visualization of the program execution by proof trees. The link between object-oriented and logic programming is realized with the help of CAPJa. Although intended for the use with PROLOG, CAPJa can also be used to integrate syntactical subsets of PROLOG, such as DATALOG or DATALOG*. With some care, the mechanisms of CAPJa work just
as well. The only difference to Prolog in our case study, is that an instance of Invoice, which has a list of positions as members, cannot be asserted as a nested term. In Datalog nested terms are not allowed. This is why we assert the positions of an invoice separately as ground terms, see also Line 10 in Listing 8.17.

In this case study about the taxes in international e-commerce, we have started the development from Java. The problem has been to develop business rules for an existing application in Java and based on existing data structures in Java, without any modifications to the original system. With CAPJa and the Jpml we can specify custom object-to-term mappings in Java. The method create of the class JPMapperFactory<T> returns a JPMapper<T> instance for which the implementation has been derived from the input lambda expression which contains the specifications of the mapping. The source code for the particular mapper is automatically generated by CAPJa. This technique requires no modification to the classes that are going to be mapped. The term structures resulting from the generated mappers provide the basic predicates for the development of the business rules which, now, can be started completely independently to the original system in Java. In this way, the development can be split in parts for an object-oriented expert and a logic programming expert without paradigmatic interference.

8.3 Validation

In this section, we first discuss the applicability of CAPJa in both case studies and illustrate further applications of CAPJa. Finally, we compare CAPJa and the related work from Chapter 3 with the criteria for a seamless integration of Prolog and Java from the object-oriented perspective that we have previously stated in Section 3.5.2.

8.3.1 Applicability Revisited

The London underground case study proves the overall applicability of the CAPJa approach. The Java interface to the Prolog knowledge base has been defined easily in Prolog with the help of the compact and declarative Psn. From it, CAPJa has generated the Java classes that form an object-oriented interface to the knowledge base in Prolog. On the Java side, object-oriented queries to Prolog have been formulated in a concise way with the help of JPQL and Boolean lambda expressions. With its declarative query format CAPJa increases the readability and the accessibility of multi-paradigm programs with Prolog and Java further. At the same time, CAPJa reduces the size of the necessary source code in Java significantly.
This improves productivity, maintainability, and refactoring. The performance tests demonstrate that the abstraction layer of CAPJa has minimal effect to the program execution. The default gateway PPG of CAPJa is a portable, alternative interface to the various PROLOG systems. The used PROLOG systems all have been tested without any modifications to the program in JAVA. The performance in the tests with the PPG turned out to be decent and sufficient for real-world applications with PROLOG and JAVA.

The case study about taxes in international e-commerce has shown that not only PROLOG but also DATALOG or DATALOG$^*$ can be integrated with and accessed from JAVA with the help of CAPJa. Coming from either direction, JAVA or PROLOG, CAPJa facilitates multi-paradigm development without paradigmatic interference. Object-oriented experts can develop independently from logic-programming experts. CAPJa provides a clean interface between both worlds. However, the case study about taxes in international e-commerce also demonstrates an interesting field for a collaborative application of PROLOG and JAVA. PROLOG can be used to keep the application logic in a compact and declarative format and separates it from the procedural, and sometimes verbose, routines in JAVA. With the help of PROLOG or DATALOG programming tools [8, 7, 29] reliable rule-based systems can be developed more easily. In particular, domain-specific languages (in PROLOG) and proof trees simplify the collaboration between programmer and domain experts.

In addition to the case studies of this chapter, we have also successfully combined PROLOG and JAVA for information extraction in XML documents and comma-separated values (Csv) files. The structure of XML documents is similar to the structure of terms in PROLOG. Each line of a Csv file that specifies an individual data record can easily be represented as relation in PROLOG. Therefore, both document types can be converted into sets of terms in PROLOG. Their signatures in PsN can be used by CAPJa for the integration into JAVA. Once integrated, we can reason in a convenient way from JAVA about the contents of a given XML document or Csv file. We have used PROLOG to compare large Csv files with over a million entries. The entries of each file had to be assigned to each other. However, the relation between the entries from the different files was not an one-to-one relation. Some entries of one file were aggregated into a single entry in the other file. Subsequently, the PROLOG program for the validation has been integrated with the help of CAPJa into a larger verification module in JAVA.

Games engineering is another interesting area for the cooperation of JAVA and PROLOG. A recent trend in the gaming industry is the virtualization of traditional board games. One advantage of virtualized board games is that the set-up time for the physical components is omitted which saves up to 30 minutes for the setup of com-
plex games. Another advantage is that virtualized board games can be accessed across networks such as the world wide web. This allows players to participate from all over the world. Because every virtualized board game has some sort of state, gaming sessions can be interrupted at any time and continued on a latter occasion. This is especially important for lengthy board games that require several gaming sessions. We have used PROLOG and JAVA for the virtualization of traditional board games. We use JAVA to represent all physical components of a board game and their visualization. Moreover, the state of a gaming session is also handled by JAVA. Board games can have a complex set of rules that have to be memorized by players. Often, we can easily transform these rules in natural language to declarative rules in PROLOG. In addition, PROLOG can be used to implement an artificial intelligence (AI) which allows us to automate the application of rules or allows players to solo multiplayer games against the AI. These components for a virtualized board game can be developed almost independently from each other. CAPJa supports a clean separation between PROLOG and JAVA, and thus allows for a clean separation in areas of responsibility. In most recent works, we have successfully implemented with CAPJa a virtual card game and a virtual dungeon crawler game\footnote{A game where heroes face fierce monsters while navigating through a labyrinthine environment.}, both based on the cooperation of PROLOG and JAVA.

8.3.2 Important Criteria for a Seamless Integration Revisited

In Section 3.5.2 we have derived from related work, and our own experiences, important criteria for a seamless integration of PROLOG and JAVA. In this section we compare our approach with CAPJa and the previously stated criteria. In the same process, we consider also the approaches of Chapter 3.

[A] Accessibility

[A1] Concise, object-oriented queries from Java to Prolog.

With CAPJa, queries from JAVA to PROLOG are expressed in JPQL, an eDSL in JAVA. JPQL allows for query constraints in the form of a Boolean lambda expression together with conditional and relational operators in JAVA. In so doing we obtain object-oriented queries that are clear and concise. The query type implicitly associates a predicate in PROLOG and thus the predicate can be queried without any knowledge in PROLOG. Solutions due to unification in PROLOG will be returned as new instances of the query type. No manual translations of variable bindings in PROLOG to JAVA data structures are necessary. INTERPROLOG \cite{15}, JPL \cite{89}, or P@J \cite{21} represent PROLOG goals in JAVA either by plain PROLOG code in string format or
by complex nested objects that represent the PROLOG data structures. All these approaches require profound knowledge in PROLOG for the creation of queries. From an object-oriented perspective, a complex blend of language artifacts in PROLOG and JAVA is less accessible and difficult to read. With LOGICOBJECTS [17], we need symbiotic methods that have to be implemented in LOGTALK [53]. In contrast to the query mechanism of CAPJa, the symbiotic methods of LOGICOBJECTS are less flexible. Moreover, the splitting of symbiotic methods in JAVA and corresponding implementations in LOGTALK threatens transparency in JAVA.


In CAPJa, the integration of PROLOG predicates into JAVA and the query mechanism for PROLOG in JAVA are based on standard techniques in PROLOG and JAVA. For the greater part, they have been automated and only the most indispensable specifications are left to the programmer. All those specifications have clear and concise formats in PROLOG or JAVA. Small domain-specific languages embedded in PROLOG and JAVA provide convenient access to the appropriate mechanisms of CAPJa. In so doing the complexity of multi-paradigm programs with PROLOG and JAVA has been further reduced. Conversely, former approaches such as INTERPROLOG, JPL, or P@J do not rely on automation. For the development in JAVA, they require profound knowledge in PROLOG. For the development with LOGICOBJECTS, a solid understanding of LOGTALK is vital.

[A3] Non-commercial, open source license.

Our current efforts are to distribute CAPJa as free plugins for the most common IDEs. We start with a free plugin for the Eclipse IDE [27]. In addition, we will soon share the sources of CAPJa via GitHub [35]. All discussed former approaches have been made publicly available, too. However, none of them support modern IDEs.

[C] Compatibility.

[C1] No dependency on a particular Prolog system.

With CAPJa, the JAVA source code for the interaction with PROLOG, e. g., for queries, is completely independent from a given PROLOG system which is abstracted by the JAVA interface JPGateway. The default gateway implementation PPG of CAPJa is a subclass of JPGateway and a portable approach which is not specific to a given PROLOG system. It has been successfully tested with several different PROLOG implementations and provides a decent performance. Alternative implementations of JPGateway, such as for remote procedure calls to PROLOG or based on particular JAVA-PROLOG bridges, can be found in Section 7.4. Almost all former
approaches only address a single particular Prolog system. The only exception is the LogicObjects approach which provides a driver-based interface to access different Prolog systems. There are only two drivers available that have been built on top of JPL and Interprolog for the Prolog systems Swi-Prolog and XSB, respectively. Moreover, the LogicObjects approach is limited to Prolog systems which have compatibility with Logtalk.

[C2] No modifications or extensions to the Java Virtual Machine.

In contrast to Logic Java, programs developed with CAPJa require no Java virtual machine with specific modifications or extensions which compromise portability. With CAPJa, an extended compiling process in Java is necessary to analyze predicate annotations, queries with JPQL, and mapper definitions with JPML. The results of this analysis are used to generate the corresponding classes that implement object-to-term mappings and queries from Java to Prolog.

[C3] No translation of Prolog programs to Java.

In contrast to Prolog Cafe, CAPJa does not translate Prolog programs to Java. Instead, we use the PSN and a mechanism based on source code generation to create an object-oriented interface in Java that is associated with the Prolog predicates characterized in PSN. However, the implementation of a predicate is not translated into Java. Therefore, CAPJa also omits the issues with built-in predicates as discussed in Section 3.5.1.

[C4] No dependency on native code.

In contrast to JPL, CAPJa contains no native code which is hardware or operating system dependent. CAPJa is completely implemented in Java and all the binaries can be easily integrated as Jar into any Java project. Moreover, on the Prolog side there is no adaptation necessary at all.

[E] (Programming) Effort

[E1] Simplified integration of Prolog predicates into Java.

The integration of a Prolog knowledge base into Java is an easy task with CAPJa. It is comparable to creating an XML Schema or a DTD for XML documents but simpler and faster. CAPJa offers the Predicate-Signature Notation (PSN) in Prolog to characterize the signatures of predicates. From predicate signatures, CAPJa automatically generates an object-oriented interface in Java that is associated with the predicates characterized in PSN. In contrast to CAPJa, only Prolog Cafe [5] provides an integration mechanism for Prolog predicates. However, the integration
features of Prolog Cafe are limited. For instance, built-in predicates of a given Prolog system that are not implemented in Prolog cannot be translated by Prolog Cafe.


Once an object-to-term mapping has been characterized with either the PSN, the @JPMapping annotation layer, or the JPL, CAPJa generates the corresponding mapper classes that automate the object-to-term conversion, and vice versa. CAPJa automatically applies its default mapping to objects in queries and to results from Prolog. Only custom mappings have to be made explicitly by a reference to a corresponding mapper. With the help of JPML, almost any Java object can automatically be mapped to a suitable predicate in Prolog, and back. INterProlog and JPL have no automated object-to-term conversion, and thus, they allow no objects as query parameters. They only support the conversion of primitive types and some collection types. LogicObjects only provides an automated conversion for instances of symbiotic classes, i.e. abstract Java classes which have their implementations reside in Logtalk.

[F] Flexibility


Next to CAPJa’s default mapping mechanism, there are two options to define custom object-to-term mappings in Java. The first option is to annotate the relevant class with a @JPMapping annotation. For the representation of class instances as terms in Prolog, the @JPMapping annotation allows us to specify the functor, arity, argument order, and structure of class fields as arguments in Prolog. The second option are custom mapping definitions with the JPML, an eDSL in Java. The JPML offers the same options as the annotation-based approach but is also applicable for Java types for which the source code is not accessible. Because P@J, Logic Java, and LogicObjects heavily rely on source code annotations, their mapping mechanisms are not applicable to Java types for which the the source code is not accessible or available.

[F2] Support for non-determinism.

Next to retrieving an individual solution or all solutions in one go, CAPJa enables the lazy evaluations of non-deterministic queries to Prolog. The method getLazySolutionsIterator of JPQuery<T> returns an iterator that allows us to traverse all solutions found by Prolog. The specialty about this iterator is that solutions are lazily return by Prolog due to controlled backtracking. In this way,
we control the backtracking in PROLOG from JAVA. Except for PDT Connector [79], PBR4J [68], and LOGICOBJECTS, approaches with low-level access to PROLOG provide similar control mechanisms for backtracking in PROLOG.

[F3] **Context-dependent return types of queries to Prolog in Java.**

With CAPJa, the context of queries to PROLOG is determined by the query type. A query type is a common JAVA type. For instance, if we query for persons with the `Person` type in JAVA, we get persons returned from PROLOG in the form of objects with `Person` type. With CAPJa, solutions returned from PROLOG are automatically converted to instances of the query type. In this way, queries with CAPJa are highly flexible and have context-dependent typed return values in JAVA. In contrast, the return of queries to PROLOG with JPL are always key-value pairs of PROLOG variable bindings. A conversion to meaningful objects in JAVA is left to the programmer. The same is true for INTERPROLOG, the return type of queries to PROLOG is `Object`, the most general type in JAVA. Queries with LOGICOBJECTS are based on (symbiotic) methods in JAVA which always have a fixed return type. However, methods with fixed return types are less flexible than the generic `JPQuery<T>` class of CAPJa with the query type T.

[O] **Obfuscation**

[O1] *No boilerplate code for queries to Prolog in Java.*

CAPJa integrates PROLOG predicates into JAVA via an object-oriented interface that associates PROLOG predicates in Psn. Together, with an automated translation of objects to terms, and vice versa, plain JAVA objects can be used in queries to PROLOG. The JPQL offers a clean and concise query mechanism in JAVA which exploits Boolean lambda expression together with conditional and relational operators. All in all, the necessary JAVA code for queries to PROLOG has been reduced to an absolute minimum with CAPJa. The only construct in CAPJa that reveals an interaction with PROLOG is the `JPQuery<T>` class and its particular methods for obtaining solutions. In contrast to JPL or P@J, no boilerplate code obfuscates the actual intention of querying PROLOG.


The mechanisms of CAPJa establish links between classes in JAVA and predicates in PROLOG. Therefore, objects in JAVA can be used unmodified for queries to PROLOG. The JPQL allows the definition of query constraints for the query type based on Boolean lambda expressions. The query type becomes the lambda parameter and can be referenced by a template. Properties of the query type can be addressed
naturally via the template; e. g., through dedicated methods. In this way, CAPJa completely preserves the look and feel of object-oriented programming in Java. Query mechanisms that utilize PROLOG code in string format such as INTERPROLOG or P@J are not object-oriented. In addition, they are error prone and difficult to debug. Programming tools such as autocompletion and syntax highlighting are also not applicable for strings. Although the term construction mechanism of JPL is object-oriented, JAVA objects cannot be directly used for queries to PROLOG. The programmer has to translate objects first to corresponding, object-oriented representations of PROLOG terms which are difficult to read and write.

[O3] Clean separation between programming in PROLOG and JAVA.

With CAPJa, almost no PROLOG language artifacts are used in JAVA. PROLOG is implicitly referenced via the generated object-oriented interface in JAVA that is associated with the PROLOG predicates characterized in PSL. All mechanism of CAPJa are implemented in JAVA and only affect the JAVA side, the development in PROLOG is completely untouched. In this way, CAPJa supports independent software development in PROLOG and JAVA. All the discussed former approaches provide no clean separation between PROLOG and JAVA, be it plain PROLOG code in string format in JAVA, PROLOG data structures in class format, or PROLOG specific annotations for queries.

[P] Productivity

[P1] Decent execution times for the routines in Java.

Custom JAVA-PROLOG interfaces particular to specific PROLOG systems are always unparalleled in performance as they usually have raw access to the internals of the PROLOG system. However, two main goals of CAPJa are accessibility and portability. The first goal has been achieved by abstraction and automation. Nevertheless, abstraction and automation are usually accompanied by a certain degree of overhead which affects the run performance. The requirements in JAVA vary considerable for each custom JAVA-PROLOG interface. To achieve the second goal, core components of CAPJa should not rely on the specifics of a particular PROLOG system or be associated with custom JAVA-PROLOG interface. In order to accomplish both design goals without affecting the performance to much, CAPJa heavily relies on code generation as part of an extended build process in JAVA. In this way, costly analyses via reflection or byte code instrumentation at runtime have been completely avoided. A system of gateways keeps queries with CAPJa independent from the specifics of particular PROLOG systems. Moreover, the default gateway PPG offers access to various PROLOG systems with decent performance. If performance really matters,
8.3 Validation

PROLOG systems can be accessed by CAPJa through custom gateways that exploit available JAVA-PROLOG interfaces. Former approaches have often been opted for a single particular PROLOG system. Only LOGICOBJECTS provides a driver-based approach to interface different PROLOG systems (SWI-PROLOG, XSB). However, the abstraction layer of LOGICOBJECTS leads to significant performance issues. In experiments, LOGICOBJECTS has been up to 23 times slower than JPL. In contrast to LOGICOBJECTS, CAPJa has performed much better. Using the JPLGateway, CAPJa has only been 10% to 30% slower than JPL.

[P2] Autocompletion and syntax highlighting for queries from Java to Prolog.

Autocompletion and syntax highlighting are today essential programming tools. Because CAPJa provides an object-oriented interface to PROLOG, autocompletion and syntax highlighting applies to all components of CAPJa. Both increase the productivity and ensures also a correct and guided usage of CAPJa’s mechanisms. Approaches that rely on PROLOG in string-format such as INTERPROLOG or PDT Connector for the queries or P@J for logic theorems, do not benefit from autocompletion and syntax highlighting.

[P3] At compile time verifiable source code for queries from Java to Prolog.

The compiler ensures type safety in JAVA and thus reduces programming errors. However, strings cannot be verified by the JAVA compiler. Therefore, CAPJa only uses strings in very few cases and only for proper names such as predicate names. However, the annotation layer of LOGICOBJECTS for symbiotic methods uses strings also for macro expressions and because queries with INTERPROLOG and PDT Connector to PROLOG are expressed in string-format, they cannot be verified by the JAVA compiler.


Depending on the used gateway, CAPJa forwards errors that occur in PROLOG to JAVA. The default gateway of CAPJa, the PPG, throws, e. g., for unknown procedure calls in PROLOG, runtime exceptions in JAVA which forward a PROLOG error message if available. Most former approaches have similar mechanics.
9 Conclusion

This chapter concludes this thesis with a summary in Section 9.1 and an outline of conceivable future work in Section 9.2.

9.1 Summary

The subject of this thesis was the investigation of novel mechanisms for the seamless integration of PROLOG and JAVA. In particular, we were interested in an object-oriented, uniform approach which is not specific to just one PROLOG system and provides convenient access to PROLOG in JAVA.

In Chapter 2, we began with introductions to the logic and the object-oriented programming paradigm as well as to the programming languages PROLOG and JAVA. For both programming languages, we discussed intriguing features, flaws, and typical application areas. The chapter was concluded by a discussion on the synergy of PROLOG and JAVA.

In Chapter 3, we discussed important related works on the integration of PROLOG and JAVA which we had divided into three integration techniques: translation of PROLOG into JAVA, embedding of PROLOG into JAVA, and communication interfaces between PROLOG and JAVA. As a result, we extracted important criteria for a seamless integration of PROLOG and JAVA from the object-oriented perspective.

In Chapter 4, we gave an overview on the Connector Architecture for Prolog and Java (CAPJa) which is an object-oriented, uniform approach to the integration of PROLOG and JAVA and the main contribution of this thesis. The integration framework CAPJa is completely implemented in JAVA and imposes no modifications to the JAVA Virtual Machine or PROLOG. In JAVA, CAPJa provides clear and concise access to PROLOG and simplifies the integration of predicates in PROLOG. All mechanisms of CAPJa are not specific to just one PROLOG system. The main components of CAPJa are JPMAPPING, JPLAMBDA, and JPGATEWAY and were discussed in the following chapters.

In Chapter 5, we presented the JPMAPPING component of CAPJa which provides an automated object-to-term mapping mechanism. In JAVA, individual mappings
can be customized with the help of a concise annotation layer. In PROLOG, object-to-term mappings can be defined in the *Predicate-Signature Notation* (**PSN**). The **PSN** allows us to characterize the signatures of PROLOG predicates in a clear and concise manner. From the predicate signatures in **PSN**, JPMAPPING generates JAVA classes that provide an object-oriented interface that is associated with the PROLOG predicates which have been characterized in **PSN**.

In Chapter 6, we introduced the JPLAMBDa component of CAPJa which provides two embedded domain-specific languages (**eDSL**) in JAVA. The *Java-Prolog Query Language* (**JPQL**) is the first **eDSL** and allows us to formulate queries to PROLOG. For this purpose, the type parameter of the generic **JQuery<T>** class is used to determine the query type in JAVA. The query type and its members implicitly reference a predicate and its arguments in PROLOG. The **JPQL** supports the definition of query constraints based on the members of the query type. They can be formulated as Boolean lambda expression. Query constraints with the JAVA equality operator lead to a substitution of the logical variables that represent the members of the query type in PROLOG. Query constraints with relational JAVA operators are translated to subgoals with the logical variables of the actual query. Multiple query constraints are connected by conditional JAVA operators and precedence is simply expressed by round brackets. In this way, queries in **JPQL** are very clear and concise.

The *Java-Prolog Mapping Language* (**JPML**) is the second **eDSL** in JAVA and can be used as an alternative to the annotation layer of the JPMAPPING component. The advantage of the **JPML** is that we can explicitly define custom object-to-term mappings, even for JAVA types for which the source code is not accessible. For the specifications of mappings, the **JPML** also uses lambda expressions in JAVA.

In Chapter 7, we discussed the JPGATEWAY component which is an extensible system of gateways to PROLOG and separates the other components of CAPJa from the specifics of particular PROLOG systems. A gateway can be used to encapsulate a particular PROLOG system. However, the *Portable Prolog Gateway* (**PPG**) is designed to work with various PROLOG systems and uses standard streams for input and output in order to communicate with a PROLOG top-level. The **PPG** has been successfully used to connect CAPJa with various PROLOG systems. CAPJa’s system of gateways can also be easily extended by custom gateways. We discussed several approaches for custom gateways, for instance for calling a remote PROLOG engines or built on top of already established JAVA-PROLOG interfaces.

In Chapter 8, we evaluated our approach with CAPJa with the help of two detailed case studies. In the first case study, CAPJa was used to integrate a PROLOG knowledge base for the network of the London underground into JAVA. The **PSN** simplified the integration efforts considerably. Only seven source lines of code (**LoC**) in PRO-
LOG were necessary to define and generate in JAVA an object-oriented interface to the knowledge base in PROLOG. All three queries of the case study were exceptionally compact in JPQL and easy to read. Compared to former integration approaches with an object-oriented term construction mechanism in JAVA, the reduction in LoC with CAPJa was significant, without affecting the implementations on the PROLOG side. Compared to the seven LoC in PROLOG, we had to implement 47 LoC in JAVA for an alternative implementation with JPL, just to create the same object-oriented interface to knowledge base in PROLOG. Moreover, with JPL we required at least 66% more LoC in JAVA for the queries to the PROLOG knowledge than with CAPJa. For the most complex query of the case study, we even required 283% more LoC with JPL than with CAPJa. The necessary implementation effort with CAPJa scaled very well with the increasing complexity of the queries in the case study. With CAPJa, all three queries required only six LoC in JAVA. Successful tests with different PROLOG implementations such as BPROLOG, the CIAO system, GNU-PROLOG, SWI-PROLOG, TU-PROLOG, the XSB system, and YAPROLOG demonstrated the portability of the PPG. The simplest query was executed via the PPG on all tested PROLOG systems in less than 1 millisecond on average. The most complex query of the case study was executed via the PPG in the best case in less than 1 millisecond on average (YAPROLOG) and in the worst case in less than 5.52 milliseconds on average (TU-PROLOG) which is still a decent result.

In the second case study, CAPJa was used to extend JAVA by business rules in DATALOG\(^*\) which is an extension to standard DATALOG and allows for a larger set of connectives (including conjunction and disjunction), function symbols, and stratified PROLOG predicates (including aggregation and default negation) in rule bodies. DATALOG\(^*\) programs are evaluated bottom-up, just like in standard DATALOG. In the case study, we derived from already defined JAVA types suitable predicate signatures that define the structure of the fact base for DATALOG\(^*\). The business rules in DATALOG\(^*\) were implemented on top of the fact base which was derived from the given JAVA types. Once the object-oriented interface to DATALOG\(^*\) was determined, the business rules were developed independently from JAVA. Objects in JAVA provided the facts necessary for the evaluation of the business rules in DATALOG\(^*\) at runtime. To bridge the gap between business analysts and programmers, the business rules were designed in a compact and declarative format. In addition, the processing of the business rules in DATALOG\(^*\) was visualized by proof trees.

We finally concluded the chapter with a discussion on the overall applicability of CAPJa and a comparison of CAPJa and the discussed integration approaches of Chapter 3 with the important criteria of Section 3.5.2 for a seamless integration of PROLOG and JAVA from the object-oriented perspective. In contrast to the previously discussed integration approaches, only CAPJa met all the stated criteria.
To this end, we hope that CAPJa will help us to better exploit decades of logic programming research from Java. Moreover, we hope that our approach will also encourage Java developers to consider declarative, rule-based problem solving in Prolog as a viable alternative.

9.2 Future Work

In the following, we outline future work that includes improvements and novel extensions to CAPJa, further case studies, and an interesting application of CAPJa.

**Publishing.** Our next step is the release of CAPJa. It will be published under an open source license and its source code will be made available through GitHub [35]. In addition, we currently develop a plugin for the Eclipse IDE which will further improve the accessibility of CAPJa’s features and increase the productivity of software development with CAPJa. We have also planned to provide CAPJa in the form of a lightweight command line tool which can easily be integrated with build automation software for Java such as Apache Ant [39].

**Gateways.** To improve the portability of CAPJa, the integrated system of gateways can be extended by more custom gateways. In particular, the integration of already available custom Java-Prolog interfaces is desirable. With regard to the PPG, more tests with more Prolog systems are necessary. Although functional, the PPG has not been opted for performance, so far. The used internal data structures as well as the integrated parsing functionality have potentials for further improvements.

**Reference Cycles.** Variables in Java follow a reference model and thus it is possible that objects contain self-references. We call this reference cycle. For instance, a Java type that represents siblings creates a reference cycle if two instances are siblings of each other. A reference cycle cannot be mapped conventionally to Prolog as this would lead to cyclic term structures. CAPJa, in its current status, does not handle reference cycles. CAPJa checks a class for possible reference cycles during the generation of a corresponding mapper via depth-first search of the class members. If such a test is positive, the usage of the mapper is discouraged in the form of a warning. It should be noted that our test does not verify the existence of a reference cycle. The existence of a reference cycle usually depends on parameters set at runtime. Therefore, a sophisticated mechanism for the treatment of reference cycle at runtime would be desirable.
Polymorphy. In object-oriented programming languages polymorphy is an important concept. In its current status, CAPJa does not support polymorphy for queries to Prolog. To be more precisely, let \( A \) be a Java type with a subtype \( B \) with corresponding representations \( a/2 \) and \( b/3 \) in Prolog. Then, we consider as support for polymorphy in our context if an associated \( a/2 \) fact can be derived for each \( b/3 \) fact and a query to Prolog with query type \( A \) in Java leads to results that have been derived either from facts \( a/2 \) or from facts \( b/3 \), without information loss. The direction from subtype to supertype of a type hierarchy can easily be represented by rules in Prolog. The following rule shows a subtype-to-supertype relation based on the predicates \( b/3 \) and \( a/2 \) that represent instances of the classes \( B \) and \( A \).

\[
a(X, Y) :- b(X, Y, _). 
\]

For \( b/3 \) facts, a query with query type \( A \) now returns additional solutions. However, the third argument of \( b/3 \) is lost in the process. To solve this issue we can represent the supertype-to-subtype relation in Prolog, too. The following rule implements this relation for the terms that represent the class instances of \( A \) and \( B \).

\[
% super(+SuperType, -SubType)
\]

\[
\text{super}(a(X, Y), b(X, Y, Z)) :-
\]

\[
a(X, Y),
\]

\[
b(X, Y, Z),
\]

\[
\text{ground}(b(X, Y, Z)).
\]

The predicate \( \text{polymorphic\_call}(+\text{Goal}, +\text{Subgoals}, -\text{Result}) \) can be queried from Java for Prolog calls with respect to the \( \text{super}/2 \) rule from above. The input argument \( \text{Goal} \) is the representation of the query type in Java as term with possible ground arguments that have been derived from equality constraints in Jpql. The input argument \( \text{Subgoals} \) is a conjunction of all relational constraints in Jpql. The output argument \( \text{Result} \) unifies with a term the either represents the query type or a subtype of the query type for which a \( \text{super}/2 \) rule is defined in Prolog.

\[
% \text{polymorphic\_call}(+\text{Goal}, +\text{Subgoals}, -\text{Result})
\]

\[
\text{polymorphic\_call}(\text{Goal}, \text{Subgoals}, \text{Result}) :-
\]

\[
\text{call}((\text{Goal}, \text{Subgoals})),
\]

\[
( \text{super}(\text{Goal}, \text{SubType}) \rightarrow \text{Result} = \text{SubType}
\]

\[
; \text{Result} = \text{Goal} ).
\]

We have not tested yet all aspects thoroughly. However, the described approach definitely leads to overhead on the Prolog side. In addition, the result handling of queries to Prolog with CAPJa has to be adapted.

Case Studies. We need more in-depth analyses of software projects with CAPJa. In particular, we are interested in case studies that evaluate user experiences with
CAPJa with regard to accessibility, the learning curve, and productivity. This is particularly important for the design of new IDE plugins with CAPJa. Based on more case studies we can also investigate how the integration of logic and object-oriented programming affects the program design. In addition, we want to discover further application areas that are suitable for the integration of PROLOG and JAVA. In an actual case study, we analyze the usage of PROLOG and JAVA for games engineering. Therefore, we have implemented the gaming logic and artificial intelligence in PROLOG and the game state and the user interface in JAVA. CAPJa has been used to connect the components in PROLOG and JAVA.

A Hybrid Relational-Deductive Database Management System in Java. An interesting further application of CAPJa is a hybrid relational-deductive database management system (hRDDBMS) in JAVA. The advantage of a hRDDBMS is that explicit knowledge in the form of entries in a relational database can be extended by implicit knowledge in the form of rules in a deductive database. In so doing data-centric applications can be supported. Most PROLOG implementations already have an abstraction layer for SQL and connectivity with relational databases, for instance via ODBC. A conceivable approach to a hRDDBMS might be the combination of CAPJa with the popular JAVA persistence framework Hibernate [77] for relational databases. We can use Hibernate’s query language HQL for queries to the connected relational database and CAPJa’s query language JPQL for queries to the connected deductive database. If facts for the deductive database have to be loaded from entries in the relational database and derived terms have to be saved to the relational database, an efficient coupling of both databases is mandatory.
Bibliography


