Semantic Technologies based on Logic Programming

by

Gergely Lukácsy

PhD Thesis

Advisor: Dr. Péter Szeredi

Department of Computer Science and Information Theory
Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics
Budapest, Hungary

Copyright © Gergely Lukácsy, 2008.
Nyilatkozat

Alulírott Lukácsy Gergely kijelentem, hogy ezt a doktori értekezést magam készítettem, és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltém.

Budapest, 2008. február 22.

Lukácsy Gergely

A disszertáció bírálatai és a védésről készült jegyzőkönyv megtekinthető a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Dékáni Hivatalában.
Abstract

This thesis describes work on how Logic Programming (LP) can be used in various knowledge intensive applications. The contributions described in the thesis include algorithms and tools for efficient Description Logics reasoning, semantic comparison of source programs and semi-automated creation of meta-information. The link between these results is that (1) they all introduce novel semantic technologies, (2) they apply logic programming techniques in general and the Prolog LP language in particular, and (3) they have been actually implemented in the form of various applications.

Semantic technologies form the basis of building software systems that work with the meaning of various kinds of information rather than with their textual form. The main idea is to reason about some sort of knowledge represented in a machine processable way. The most important parts of this process are knowledge representation and knowledge management.

The dissertation deals with two groups of knowledge representation and management technologies: Logic Programming and Description Logic systems. Combination of these approaches creates interesting challenges: the idea of the so-called Description Logic Programming has been introduced in 2003. Several results in the present dissertation contribute to this area of semantic technologies.

In the context of Description Logics we have addressed two issues. The first issue is about how to efficiently reason on Description Logic knowledge bases when huge amounts of data are present. We have created the theoretical basis of a new Description Logic reasoning system, called DLog, that transforms description logic axioms into a Prolog program. This transformation is done independently from the individuals: they are accessed dynamically during the Prolog execution of the generated program. For this we have created several algorithms and proved their useful properties. As a result, with our implementation, we reached better scalability and more efficient execution compared to that of the earlier approaches.

The second issue addresses the question of how to use conceptual models for information integration. We have created a transformation framework together with a modelling methodology that uses Prolog for querying conceptual models formulated in Description Logics. Here we apply the closed world assumption as we argue it fits the context of information integration better. We demonstrate this approach by applying it for the SINTAGMA information integration system.

The thesis also discusses two use cases that present the viability of Logic Programming for knowledge intensive applications. First, we deal with the issue of how to detect plagiarism between source programs. We have designed and implemented Match, a generic plagiarism detection framework that aims at the semantic comparison of source programs. The idea is to transform the source code into mathematical objects, use appropriate reduction and comparison methods on these, and interpret the results appropriately. We have been using the Prolog implementation of the system at the Budapest University of Technology and Economics to successfully detect plagiarism in homework assignments for the past eight years.

Finally, we investigate the question of how to deduce meta-information in systems working with various kinds of documents. We have developed the SREngine framework, which manages a pool of generic objects together with their properties and supports Prolog based reasoning on these. The idea is to infer new properties about the objects, using their existing properties and a set of user-defined rules. These rules are given in a special logic based language developed for SREngine.
Acknowledgements

First of all I would like to thank my supervisor Péter Széredi. I met Péter during the Declarative Programming course at the Budapest University of Technology and Economics in spring 2000 where he was teaching the Prolog language. On the very first lecture he said that he had been in love with Prolog for more than 25 years. At that time I did not know what Prolog was and I did not understand how somebody could love a programming language other than Pascal, C or Java.

After the course finished I contacted Péter and we started to work on our first project which resulted in a multiple-prize-winner scientific student paper. After this we have worked together on almost everything: another student paper, my master thesis, my PhD. During these years I have been working on many Hungarian and European Union research projects under Péter’s supervision. Since 2004 we have been giving a university course on Semantic Web. Building on our teaching experience we published a Hungarian textbook, which was also accepted for publication by the Cambridge University Press.

I could always rely on Péter: not only in terms of work, but also in my private life. I am indebted to Péter for introducing me to the topic of Logic Programming, for believing in me and for supporting me all over these long years. Without him I would not be even close to what I have managed to achieve.

I am also very thankful to Tamás Benkő. Similarly to Péter, Tamás was there from the beginning: he helped me with my student papers and with my other projects. I have learnt much from Tamás as a professional and as a person, e.g. his calm attitude towards people is very unique. Tamás also took part in my personal life: we were running and hiking on many-many occasions together. Since he went to live abroad I have been missing our interesting conversations.

I would like to thank the co-authors of my publications and my recent results, especially Zsolt Nagy and Zsolt Zombori who is still an active developer of the DLog project.

I would also like to thank my roommates in the PhD room, especially Bálint Laczay and György Ottucsák. We had lots of fun together, including playing chess at worktime and lots of funny every-day discussions. It is not the same since Bálint went abroad and I moved into a different room.

Finally and most naturally, I am indescribably thankful to my family for having supported my dreams since my childhood. They have always been there no matter what. All my desire is to transfer the same love to my own children.

Work described in this thesis was supported by grants from the IST subprogramme of European Union Framework Programmes 5 and 6, for projects SILK and FUSION; by the National Office for Research and Technology of Hungary, through projects LOBO and SINTAGMA, and by the Deák Ferenc scholarship from the Ministry of Education of Hungary. The support of Sense/Net Ltd. and of the Hungarian research programme GVOP 2005/3.3.3 is also acknowledged.
# Contents

## 1 Introduction
- 1.1 Preliminaries .................................................................................. 2
  - 1.1.1 Description Logics ................................................................. 2
  - 1.1.2 Logic Programming ............................................................... 2
- 1.2 Thesis overview ............................................................................. 3
  - 1.2.1 Context ................................................................................... 3
  - 1.2.2 Problem formulation ............................................................... 4
  - 1.2.3 Approach and results .............................................................. 5
- 1.3 Structure of the Thesis and contributions ......................................... 6
  - 1.3.1 Algorithms and tools for Description Logics ...................... 6
  - 1.3.2 Case studies in semantic technology applications ............ 6
- 1.4 Summary of publications ................................................................ 6

## I Algorithms and tools for Description Logics

### 2 Efficient Description Logic reasoning in Prolog
- 2.1 Introduction .................................................................................... 13
- 2.2 Background and related work ....................................................... 14
  - 2.2.1 Reasoning on DLs ................................................................... 14
  - 2.2.2 Resolution theorem proving for DLs .................................... 14
  - 2.2.3 Description Logics and Logic Programming ...................... 15
  - 2.2.4 Prolog Technology Theorem Proving: contrapositives, ancestor resolution ....... 15
- 2.3 $SHIQ$ reasoning in Prolog ............................................................ 16
  - 2.3.1 Translating by hand: two motivating examples ............... 16
  - 2.3.2 Building DL clauses from a $SHIQ$ knowledge base .......... 18
  - 2.3.3 Specialising PTTP for DL clauses ...................................... 20
  - 2.3.4 Interpreting DL predicates ................................................. 24
  - 2.3.5 Compiling DL predicates ................................................... 25
  - 2.3.6 Compilation examples .......................................................... 26
- 2.4 Optimising DL compilation .......................................................... 28
  - 2.4.1 Principles of optimisation ..................................................... 28
  - 2.4.2 Filtering ................................................................................ 30
  - 2.4.3 Classification ....................................................................... 31
  - 2.4.4 Body ordering .................................................................... 33
  - 2.4.5 Multiple argument indexing ............................................. 35
  - 2.4.6 Ground goal optimisation .................................................... 36
  - 2.4.7 Decomposition ................................................................... 37
  - 2.4.8 Projection and supersets .................................................... 39
  - 2.4.9 Transforming role axioms ................................................... 43
- 2.5 The DLog system .......................................................................... 46
  - 2.5.1 Architecture ................................................................. 46
Chapter 1

Introduction

Semantic technologies are computer based techniques and standards that aim to formalise and utilise knowledge in various fields. Semantic technologies can take different forms depending on the application areas and the exact role they serve in the specific applications. In general, however, semantic technologies are linked by the common aim to capture knowledge in a machine processable form.

Knowledge representation has a long history that started back in the 1950s when the need for symbolic calculations have emerged and thus the idea to use programming languages based on some kinds of mathematical formalism. This led to the development of declarative languages, such as LISP or Prolog.

In the 1970s the knowledge representation approaches were categorised into two main directions: (1) logic based technologies that used reasoning methods developed for the first order logic and (2) frame based systems. These systems tried to imitate the human mind by thinking in terms of classes, relations and inheritance and as such, they formed the basis of the object-oriented and Description Logic paradigms. The logical foundations of frame-based languages was introduced in the late 1970s by Pat Hayes.

The 1980s were dominated by the Japanese 5th generation project that aimed at creating intelligent, Prolog based, massively parallel computers.

In the mid 1990s, the so called Semantic Web vision started to materialise and the community has created three important standards: the Resource Description Framework (RDF), the RDF Schemas (RDFS) and the Description Logic based Web Ontology Language (OWL). The Semantic Web approach uses two fundamental ideas. One is to associate machine processable meta-information with Internet-based resources which is the main task of the RDF language. The other idea is the ability to do reasoning on the meta-information. The background knowledge required for reasoning is described in RDFS and OWL.

Parallel with this, other technologies started to make use of the results of the Semantic Web approach. Several standards emerged for semantic web services and semantic discovery, such as SAWSDL or OWL-S. Integration approaches also started to benefit from semantics: several projects aim at the semantic integration of various kinds of information sources, in most cases using Description Logic based ontologies and Semantic Web technologies.

The idea of the so called Description Logic Programming, introduced in 2003, aims at creating a new logical knowledge representation formalism that combines the common elements of Description Logics and Logic Programming. This initiative clearly shows that the rule based paradigm of Logic Programming can be useful when combined with other knowledge representation technologies.

There are several scientific conferences with long history that are related to semantic technologies. In the context of the Web we have the International and European Semantic Web conferences (ISWC and ESWC) and the World Wide Web conference series (WWW). For researches interested in knowledge representation the International Joint Conference on Artificial Intelligence (IJCAI) and the International Workshop on Description Logics (DL) serve as a forum. In the area of Logic Programming the International Conference on Logic Programming (ICLP) and several smaller conferences, such as the Practical Aspects of Declarative Languages (PADL), represent the highest standard. In the recent years several new conferences has also emerged. For example, the International Semantic Technology Conference was initiated in 2005, the first European Semantic Technology Conference was organised in Austria in 2007.

In the following we give an overview of the thesis. First, the basic ideas of Description Logics and
Logic Programming are outlined. Next, a summary of the thesis is presented showing the problems to be solved, the approach to their solution and the results achieved. Finally, the structure of the remaining part of the thesis is outlined together with the summary of the relevant publications.

1.1 Preliminaries

Below we give a brief introduction to Description Logic and Logic Programming as these technologies form the basis of the dissertation.

1.1.1 Description Logics

Description Logics (DLs) [59] is a family of simple logic languages used for knowledge representation. DLs are used for describing various kinds of knowledge of a selected field as well as of general nature. The description logic approach uses concepts to represent sets of objects, and roles to describe binary relations between concepts. Objects are the instances occurring in the modelled application field, and thus are also called instances or individuals.

A description logic knowledge base \( KB \) is a set of DL axioms consisting of two disjoint parts: the TBox and the ABox. These are sometimes referred to as \( KB_T \) and \( KB_A \). The TBox (terminology box), in its simplest form, contains terminology axioms of form \( C \sqsubseteq D \) (concept \( C \) is subsumed by \( D \)). In more complex DLs the TBox may also contain axioms about roles (see below). The ABox (assertion box) stores knowledge about the individuals in the world: a concept assertion of form \( C(i) \) denotes that \( i \) is an instance of concept \( C \), while a role assertion \( R(i,j) \) means that the objects \( i \) and \( j \) are related through role \( R \). Usually we assume that two different individual names denote two different individuals (this is the so called unique name assumption).

Concepts and roles may either be atomic (referred to by a concept name or a role name) or composite. A composite concept is built from atomic concepts using constructors. The expressiveness of a DL language depends on the constructors allowed for building composite concepts or roles. Obviously there is a trade-off between expressiveness and inference complexity.

We use the DL language SHIQ in this thesis. Here, concepts (denoted by \( C \) and \( D \)) are built from role names (denoted by \( R \) and \( S \)), concept names (also called atomic concepts), the top and bottom concepts (\( \top \) and \( \bot \)) using the following constructors: intersection (\( C \sqcap D \)), union (\( C \sqcup D \)), negation (\( \neg C \)), value restriction (\( \forall R.C \)), existential restriction (\( \exists R.C \)), and qualified number restrictions (\( \geq n R.C \) and \( \leq n R.C \)).

In addition to the concept subsumption axioms introduced above, one can also define a hierarchy between roles (\( R \sqsubseteq S \)), use the inverse of a role (\( R^{-} \)) and finally, declare that a role is transitive (\( \text{Trans}(R) \)). The role hierarchy and transitivity axioms are sometimes referred to as the RBox (role-box). An important sublanguage of SHIQ is \( ALCN \), where we only have a limited form of number restrictions (\( \geq n R \) and \( \leq n R \)) and do not allow any role related constructs. DL languages can also be extended with datatypes, such as integers or strings, when building concepts. This is usually denoted by a \( (D) \) after the language name, e.g. \( ALCN(D) \). For more details we refer the reader to [4].

The basic inference tasks concerning the TBox can be reduced to determining if a given concept \( C \) is satisfiable with respect to a given TBox or not.

ABox-inference tasks require both a TBox and an ABox. In this thesis, we will deal with two ABox-reasoning problems: instance check and instance retrieval. In an instance check problem, a query-concept \( C \) and an individual \( i \) is given. The question is whether \( C(i) \) is entailed by the TBox and the ABox. In an instance retrieval problem the task is to retrieve all the instances of a query-concept \( C \), entailed by the TBox and the ABox.

1.1.2 Logic Programming

The main idea of Logic Programming is to use mathematical logic as a programming language. The execution of a logic program can be viewed as a reasoning process.
Prolog (Programming in Logic) [115] is the first and so far the most widely used logic programming language. Prolog uses Horn-clauses and SLD resolution [32] for reasoning. The basic elements of the Prolog execution process are procedure invocation based on unification and backtracking [123].

Prolog, and logic programming in general, is successfully used in several areas of computer science. These include natural language processing, planning, expert systems, different kinds of reasoning systems, and information integration.

The notion of term is a principal concept of the Prolog language. It is either (a) a simple value (number, string) or (b) a variable or (c) a structure with a name and arbitrary number of arguments. These arguments are Prolog terms themselves. The name and the arity of a term together is referred as the functor of the term. A Prolog structure with three arguments can be seen below:

\begin{equation}
\text{person(}\text{Name}, \{\text{A, B, C, D, E}\}, \_\text{)}
\end{equation}

Here the name of the structure is person. The first and the third arguments are variables. These are denoted by identifiers starting with a capital or an underline. A single underline (\_) is an anonymous variable, the value of which is of no interest. Multiple occurrences of such anonymous variables are considered different. The second argument of (1.1) is a structure in a special list notation. A list is actually a recursive structure \([\text{Head}|\text{Tail}]\), consisting of a \text{Head} (its first element) and a \text{Tail}, which is a list of the remaining elements. The list in the second argument contains five variables and is given in a simplified notation, i.e. \([\text{A, B, C, D, E}]\), in fact, corresponds to \([\text{A}|\text{B}|\text{C}|\text{D}|\text{E}||\text{]})]. Here [] represents an empty list (a list with no elements).

A Prolog program consists of a set of clauses of form \text{Head} :- \text{Body}, meaning \text{Head} is implied by \text{Body}. The \text{Head} is a term, while the \text{Body} is a term or a comma-separated sequence of terms. Here the comma denotes a conjunction. Clauses whose heads have the same functor are grouped together into predicates. The name of a predicate is the shared structure name of the heads of its clauses.

A Prolog goal (query) has the same form as a clause body. The execution of a goal wrt. a Prolog program succeeds if an instance of the goal can be deduced from the given program (not shown here). A goal can succeed multiple times, providing different variable substitutions as results. Let us consider the goal shown below.

\begin{align*}
\text{writer(}\text{ID}), \text{painter(}\text{ID})
\end{align*}

This complex goal consists of two goals, separated by a comma. It succeeds if there is such an instantiation of variable \text{ID} under which both goals can be deduced from the given program (not shown here). The result of the execution is the enumeration of such \text{IDs}. Informally, this query enumerates those people who are writers and painters at the same time.

Further control constructs such as disjunction (\text{Goal1 ; Goal2}) and negation \text{\textbackslash +Goal} are also supported by Prolog. The latter is the so called “negation by failure”, which is not capable of enumerating solutions, but just checks if the execution of \text{Goal} fails. The Prolog language also has a wide range of built-in predicates.

More about Prolog can be read in the ISO standard for Prolog [115] and in textbooks, such as [123, 23].

1.2 Thesis overview

This section presents an overview of the thesis. First we describe the context of the thesis. Next we summarise the research questions and then discuss the approach to their solutions and the results achieved.

1.2.1 Context

I have been working on various research projects involving semantic technologies and Logic Programming since 2000. My first project was about the semantic comparison of source programs, which led to the development of a plagiarism detection system for comparison of the student homework assignments submitted for the course on Declarative Programming at the Budapest University of Technology and Economics. This work resulted in a student conference paper that won first prizes both on university and on national level [83]. This forms the basis of Chapter 4 in the present dissertation.
In early 2002 I joined the SILK (System Integration via Logic and Knowledge) international project [74] supported by the IST subprogram of the 5th Framework EU research program. The aim of the SILK project was to build an Enterprise Information Integration system using knowledge representation and reasoning techniques to support both mediation and integration of heterogeneous information sources. My role in the project was to extend the capabilities of SILK towards managing Semantic Web sources: I developed the RDF-wrapper of SILK, which not only gave access to semantic information available in RDF format, but also included its own reasoning and consistency checking capabilities [12].

The work started in SILK was continued in the LOBO (LOgic Based management of Ontologies) project in 2003–2004. LOBO was a Hungarian IKTA research project with the aim of using innovative ontology management techniques in information integration and applying it in the medical domain [128, 130, 129, 131, 134, 133, 135, 132]. Here I developed a robust ontology editor that was able to manage very large medical ontologies. I also contributed to the development of a logic based solution for automatic classification of patient health records [67].

SINTAGMA (Semantic Integration Technology Applied in Grid-based, Model-driven Architectures) was an NKFP-supported Hungarian national project which run between 2005–2007, and which built on the results of SILK and LOBO [78, 77, 75, 79, 76]. The main goal of the project was the development and experimental application of a novel information integration technology which includes conceptual level integration and is based on virtual database management in distributed and grid architectures. My main contributions here included the design of the architecture of the system, the development of the web based interface of SINTAGMA and the Description Logic extension of the system. This forms the basis of Chapter 3 of the present dissertation.

Since 2005 I have been involved in the DLog project which aims at developing efficient Description Logic reasoning algorithms for large datasets. This work was actually also motivated by the SINTAGMA framework, where we use Description Logic reasoning over databases containing huge amounts of data. In DLog, unlike SINTAGMA, we use the open world assumption which requires a fundamentally different approach to what we applied in SINTAGMA. I am the lead architect and developer of the DLog system and my work here forms the basis of Chapter 2 of this dissertation.

In 2006 I have participated in a Hungarian GVOP project, the aim of which was to add reasoning capabilities to an existing document management framework. My contribution here was the design and development of the SREngine system which forms the basis of Chapter 5 of the present dissertation.

Since 2006 I have been participating in the FUSION FP6 EU project [40, 41]. FUSION promotes efficient business collaboration and interconnection between enterprises by developing a framework for the semantic fusion of heterogeneous service-oriented business applications. My contribution to FUSION includes knowledge transfer of various semantic technologies used in the project.

### 1.2.2 Problem formulation

The overall goal of the work described in this thesis is to

prove the viability of Logic Programming as a paradigm for declarative knowledge representation and reasoning for semantic technologies.

This goal is achieved by presenting several use cases where using Logic Programming ideas turn out to be the key elements of the solution. We now discuss the specific issues that have emerged within these use cases.

**Problem I: Traditional Description Logic reasoning is infeasible for large data sets**

Traditional algorithms for Description Logic instance retrieval are inefficient for large amounts of underlying data. The fundamental reason behind this is that for querying description logic concepts these algorithms need to examine the whole content of the database which results in serious scalability problems. As Description Logics are becoming popular in areas such as the Semantic Web and information integration, it is very important to have systems that can reason efficiently over large data sets.
Problem II: UML modelling alone is not adequate for conceptual models in information integration

Information Integration systems are based on the idea to manage the models of the information sources represented in some modelling framework, such as UML or Description Logics. These models are not homogeneous as some of them represent fairly precise existing or virtual information sources, while others represent high-level mental models of user groups. Thus it is important to have a hybrid approach where different modelling paradigms can be used for models at different abstraction levels.

Problem III: Using lexical information when comparing source codes is not enough

Solutions for detecting plagiarism in essays and other kinds of written materials are not adequate when comparing source programs. It is an important task, however, to detect the duplication of programs or parts of these in case of programming assignments. Unfortunately, checking these programs by hand seems to be beyond possibility. Thus, it is important to develop methods and tools to assess the similarity of programs in order to narrow down the need for manual testing to an acceptable amount.

Problem IV: Manually providing meta-information for documents is infeasible

Document management systems attach certain kinds of meta-information to documents for organisation and query purposes. The most important piece of such meta-information is the classification of a given document. Usually, such meta-information is specified manually, but this may require enormous efforts. It is also true that some kinds of meta-information, e.g. the number of references to the document, cannot be managed manually as they change over time. Thus it is important to have tools and methods that allow the attachment of meta-information to documents in a more automated way.

1.2.3 Approach and results

We now discuss how the problems formulated in the previous section were approached, and what solutions we managed to achieve.

Solution to problem I: Resolution based two-phase Description Logic reasoning

I have developed an approach to transform description logic axioms into a Prolog program. This transformation is done without any knowledge on the particular individuals: they are accessed dynamically during the normal Prolog execution of the generated program. This technique, together with the top-down Prolog execution, implies that only those pieces of data are accessed which are indeed important for answering the query. This allows us to store the individuals in a database instead of memory, which results in better scalability and helps using description logic ontologies directly on top of existing information sources.

Solution to problem II: Hybrid modelling methodology

I have extended the modelling language of an existing information integration system to incorporate several Description Logic constructs, in addition to the UML-like ones. In the envisioned scenario, the high-level mental models of the users are formulated in Description Logic and via appropriate definitions they are connected to the more precise object oriented models. I developed the corresponding modelling methodology and specified the ways how Description Logic constructs are transformed into query goals.

Solution to problem III: Structural comparison of source codes

I have designed a generic program structure comparison framework and implemented it for the Prolog and SML programming languages. The main idea is to transform the source codes into mathematical objects. Now, instead of providing sophisticated comparison techniques that are resistant to the most common tricks we apply reduction steps to create more abstract views of the programs. These views are then compared by using relatively simple comparison algorithms. This approach makes our system fairly efficient and easy to customise.
Solution to problem IV: Applying rules to infer new meta-information

I have designed and developed a framework that manages a pool of generic objects together with their properties and supports reasoning on these. The main idea of the system is to infer new properties about the objects, using their existing properties and a set of user defined rules. These rules are given in a special logic programming language, providing intuitive syntax and adequate expressive power.

1.3 Structure of the Thesis and contributions

The thesis has two main parts, both containing two chapters. The first part deals with the issue of how Logic Programming can help in solving certain Description Logics related problems. The second part of the dissertation focuses on applications where the usage of Logic Programming plays a crucial role.

The main contribution of this thesis is a better understanding of the relation between Logic Programming and Semantic Technologies. In several cases, ideas originating from Logic Programming turn out to be very useful when dealing with problems that requires the intensive usage of some kinds of knowledge. In the separate chapters of the dissertation I make a number of other contributions as well.

1.3.1 Algorithms and tools for Description Logics

Chapter 2: Efficient Description Logic reasoning in Prolog

This chapter introduces the DLog system, including the theoretical background, implementation details and performance evaluation. DLog is an efficient, resolution based, Description Logic ABox reasoner fully implemented in Prolog. My main contribution in this work is an optimised transformation process that compiles Description Logic axioms into Prolog programs.

Chapter 3: DL-based conceptual modelling in information integration

This chapter describes the conceptual modelling extension of the SINTAGMA information integration system. My contribution in this chapter is a hybrid modelling paradigm, paired with the corresponding execution mechanism, where the mixture of UML and Description Logic models form the basis of information integration.

1.3.2 Case studies in semantic technology applications

Chapter 4: Plagiarism detection in programs

This chapter introduces the Match plagiarism detection system, presenting the theoretical background, implementation specific details and performance evaluation. My main contribution here is the design and implementation of a comparison framework which uses abstraction levels and simple comparison methods to determine the similarity of source programs.

Chapter 5: Logic based management of documents

This chapter describes the SREngine system, introducing its architecture, usage scenarios and implementation details. SREngine is a generic rule-based reasoner, the primary aim of which is document classification. My contribution here is the architecture and implementation of the system and a special rule language, providing intuitive syntax and adequate expressive power.

1.4 Summary of publications

Chapter 2 on the DLog system is based on the following publications.


The results of Chapter 3 were published previously in the following papers:


Work on the Match system, described in Chapter 4, was published in the following papers:

Chapter 5 on the SREngine system is based on the following publications.


Other publications and research project deliverables.


- [77] Péter Krauth, et al., Requirements specification (in Hungarian), SINTAGMA research project deliverable E2, September 2005. My contribution: collecting the requirements of BUTE.


Part I

Algorithms and tools for Description Logics
Chapter 2

Efficient Description Logic reasoning in Prolog

Traditional algorithms for description logic (DL) instance retrieval are inefficient for large amounts of underlying data. As description logic is becoming popular in areas such as the Semantic Web and information integration, it is very important to have systems that can reason efficiently over large data sets.

In this chapter we present an approach to transform description logic axioms described in the $SHIQ$ DL language into a Prolog program. This transformation is done without any knowledge on the particular individuals: they are accessed dynamically during the normal Prolog execution of the generated program. This technique, together with the top-down Prolog execution, implies that only those pieces of data are accessed which are indeed important for answering the query. This allows us to store the individuals in a database instead of memory, which results in better scalability and helps using description logic ontologies directly on top of existing information sources.

The transformation process consists of two steps: (1) first we create FOL clauses of restricted form from the DL axioms, (2) then we generate a Prolog program from these. Step (2) actually works on more general clauses that can be obtained by applying step (1) on a $SHIQ$ knowledge base. This allows us to reason on knowledge going beyond $SHIQ$.

During the transformation we apply several optimisation steps to arrive at more efficient Prolog programs. Some of these steps are specific to our approach, while others are general enough to be interesting for implementors of description logic reasoners not using Prolog.

Based on the above ideas we have implemented in Prolog a DL reasoner, called DLog, which is available on SourceForge to download. In the chapter we evaluate the performance of DLog and compare it to the the best available description logic reasoners: RacerPro, Pellet and KAON2.

2.1 Introduction

Description Logics (DLs) are becoming widespread thanks to the recent trend of using semantics in various systems and applications. As an example, in the Semantic Web idea, semantics is captured in the form of expressive ontologies, described in the Web Ontology Language (OWL) [8]. This language is mostly based on the $SHIQ$ description logic language and it is intended to be the standard knowledge representation format of the Web. Other application fields of description logics include natural language processing [38], medical systems [124], information integration [18] and complex engineering systems [28].

Similarly to [99], the motivation for our work comes from the realisation that description logics are, or soon will be used over large amounts of data. In an information integration system, for example, really huge amounts of data are stored in external databases. On the Web, as another example, we already have huge amounts of meta-information which will significantly increase as the Semantic Web vision becomes more and more tangible. Obviously, these information sources cannot be stored directly in memory.

Thus, we are interested in querying description logic concepts where the actual data – the so called ABox – is stored in databases. We found that most existing DL reasoners are not suitable for this task,
as these are not capable of handling ABoxes stored externally. This is not a technical problem: existing algorithms for querying description logic concepts need to examine the whole ABox to answer a query which results in scalability problems and undermines the point of using databases. Because of this, we started to investigate techniques which allow the separation of the inference algorithm from the data storage.

We have developed a solution, where the inference algorithm is divided into two phases. First we create a query-plan from the actual DL knowledge base, without any knowledge of the content of the underlying data set. Subsequently, this query-plan can be executed on real data, to obtain the required results.

Naturally, the quality of the query-plan greatly affects the performance of the execution. We have applied several optimisations to make the generated Prolog program more efficient. These ideas are incorporated in the system called DLog, available at http://dlog-reasoner.sourceforge.net.

This chapter is structured as follows. In Section 2.2 we introduce Description Logics and summarise theorem proving approaches for DLs. Section 2.3 presents two motivating examples to demonstrate the non-trivial nature of the translation of description logic axioms to Prolog and describes a complete, but inefficient solution for generating Prolog programs from $SHIQ$ knowledge bases. Section 2.4 is the main contribution of the chapter. It discusses several optimisation schemes which significantly increase the efficiency of execution. Section 2.5 presents the architecture and implementation details of the DLog system. In Section 2.6 we show the performance analysis of DLog comparing it with other reasoning systems. Finally, in Section 2.7 and 2.8, we conclude with the future work and the summary of our results.

### 2.2 Background and related work

In the following we first give an overview of the traditional tableau-based DL reasoning approaches. Next, we discuss how resolution can be used for DL reasoning, and summarise related work on using Logic Programming for Description Logic reasoning including our earlier contributions. Finally, we present the Prolog Technology Theorem Proving (PTTP) approach the techniques of which are used extensively throughout the chapter.

#### 2.2.1 Reasoning on DLs

Several techniques have emerged for dealing with ABox-reasoning. Traditional ABox-reasoning is based on the tableau inference algorithm, which tries to build a model showing that a given concept is satisfiable [4]. To infer that an individual $i$ is an instance of a concept $C$, an indirect assumption $\neg C(i)$ is added to the ABox, and the tableau-algorithm is applied. If this reports inconsistency, $i$ is proved to be an instance of $C$.

The main drawback of this approach is that it cannot be directly used for high volume instance retrieval, because it would require checking each instance in the ABox one by one.

To make tableau-based reasoning more efficient on large data sets, several techniques have been developed in recent years [49]. These are used by the state-of-the-art DL reasoners, such as RacerPro [50] or Pellet [122], the two tableau reasoners used in our performance evaluation in Section 2.6.

Extreme cases involve serious restrictions on the knowledge base to ensure efficient execution with large amounts of instances. For example, [60] suggests a solution called the instance store, where the ABox is stored externally, and is accessed in a very efficient way. The drawback is that the ABox may contain only axioms of form $C(a)$, i.e. we cannot make role assertions.

#### 2.2.2 Resolution theorem proving for DLs

[62] discuss how a first order theorem prover such as Vampire can be modified and optimised for reasoning over description logic knowledge bases. This work, however, mostly focuses on TBox reasoning.

In [65], a resolution-based inference algorithm is described, which is not as sensitive to the increase of the ABox size as the tableau-based methods. The system KAON2 [99] implements this method and provides reasoning services over the description logic language $SHIQ$. In Section 2.6 we use KAON2 as one of the systems with which we compare the performance of DLog.

The main idea of KAON2 is to first transform a $SHIQ$ knowledge base into a skolemised first-order clausal form. However, instead of using direct clausification, first the structural transformation [114] is
applied on the \( KB_T \) axioms. This transformation basically eliminates the nested concept descriptions by introducing new concepts; the resulting set of first order clauses is denoted by \( \Xi(kb) \). In the next step basic superposition [106] is applied to saturate \( \Xi(kb) \). The resulting set of clauses is denoted by \( \Gamma(kb) \). Clauses \( \Gamma(kb_T) \cup \Xi(kb_T) \) are then transformed into a disjunctive datalog program [29] entailing the same set of ground facts as the initial DL knowledge base. This program is executed using a disjunctive datalog engine written specifically for KAON2. In this approach, the saturated clauses may still contain (non-nested) function symbols which are eliminated by introducing a new constant \( f_i \) standing for \( f(i) \) for each individual \( i \) in the ABox. This effectively means that KAON2 has to read the whole content of the ABox before attempting to answer any queries.

Although the motivation and goals of KAON2 are similar to ours, unlike KAON2 (1) we use a pure two-phase reasoning approach (i.e. the ABox is not involved in the first phase) and (2) we translate into Prolog which has well-established, efficient and robust implementations. More details are provided in the upcoming sections.

2.2.3 Description Logics and Logic Programming

[46] introduces the term Description Logic Programming. This idea uses a direct transformation of \( \mathcal{ALC} \) description logic concepts into definite Horn-clauses, and poses some restrictions on the form of the knowledge base, which disallow axioms requiring disjunctive reasoning. As an extension, [46] introduces a fragment of the \( S\mathcal{HIQ} \) language that can be transformed into Horn-clauses. This work, however, still poses restrictions on the use of disjunctions.

Another approach of using Logic Programming in DL reasoning was proposed by the research group of the authors of the present dissertation. Earlier results of this work have been published in several conference papers. The first step of our research process resulted in a resolution-based transformation of ABox-reasoning problems to Prolog for the DL language \( \mathcal{ALC} \) and an empty TBox [104]. The second step involved examining how ABox-reasoning services can be provided with respect to a non-empty TBox: we have developed an extension able to handle restricted \( \mathcal{ALC} \) TBox-axioms as well when performing ABox-inference [103]. In [89] we presented a system doing almost full \( \mathcal{ALC} \) reasoning using an interpreter applying PTTP techniques (see Section 2.2.4 below).

Zsolt Zombori, a member of our research group, has extended the saturation technique of [99] so that the function symbols are eliminated from the resulting first-order clauses [142]. The basic idea here is to use a slightly modified version of the basic superposition where the order of certain resolution steps is changed. [142] showed that these modifications do not affect satisfiability and they require a finite number of additional inference steps compared to the “standard” basic superposition.

2.2.4 Prolog Technology Theorem Proving: contrapositives, ancestor resolution

The Prolog Technology Theorem Prover approach (PTTP) was suggested by Mark E. Stickel in the late 1980’s [126]. PTTP is a sound and complete approach that builds a first order theorem prover on top of Prolog. This means that an arbitrary set of general clauses can be transformed into a set of Horn-clauses and Prolog execution performs first order logic reasoning.

In PTTP each first order clause gives rise to a number of Horn-clauses, the so-called contrapositives. A FOL clause \( \bigwedge_{1 \leq i \leq n} \neg L_i \) has \( n \) contrapositives of form \( \neg L_k : \neg L_1, \ldots, \neg L_{k-1}, \neg L_{k+1}, \ldots, \neg L_n \), for each \( 1 \leq k \leq n \), where \( L \) is a literal (negated or non-negated atomic predicate). Having removed double negations, the remaining negations are eliminated by introducing new predicate names for negated literals. For each predicate name \( P \) a new predicate name \( \text{not}\_P \) is added, and all occurrences of \( \neg P(X) \) are replaced by \( \text{not}\_P(X) \), both in the head and in the body. The link between the separate predicates \( P \) and \( \text{not}\_P \) is created by ancestor resolution, see below.

Note that by using contrapositives each literal of a FOL clause will appear in the head of a Horn clause, ensuring that it can participate in a resolution step in spite of the restricted selection rule of Prolog.

In the PTTP approach ancestor resolution is used to support the factoring inference rule: two unifiable literals in the resolvent can be replaced by a single one [73]. Ancestor resolution is implemented in Prolog by building an ancestor list which contains open predicate calls (i.e. calls which were entered or re-entered, but have not been exited yet, according to the Procedure-Box model [108]). If the ancestor list contains a
literal which can be unified with the negation of the current goal literal, then the goal literal succeeds and the unification with the ancestor element is performed. Note that in order to retain completeness, as an alternative to ancestor resolution, one has to try to prove the current goal using normal resolution.

There are two further features to make the PTTP approach complete. First, to avoid infinite loops, iterative deepening is used as an alternative of the standard depth-first Prolog search strategy. Second, in contrast with Prolog, PTTP uses occurs check during unification.

As a summary, PTTP uses five techniques to build a first order theorem prover on the top of Prolog: contrapositives, new negated literals, ancestor resolution, iterative deepening and occurs check.

2.3 **SHIQ reasoning in Prolog**

In this section we deal with the transformation of SHIQ knowledge bases into Prolog and the execution of such Prolog code. Following two motivating examples, we summarise the principles of the SHIQ transformation. Next, we introduce the techniques—in part borrowed from PTTP—that are needed to execute the Prolog code resulting from the transformation, and present a simple interpreter for this task. Subsequently we describe how to extend the transformation process, so that the code can be directly executed by Prolog, thus resulting in a SHIQ to Prolog compiler. Finally, we show examples of this complete transformation process.

2.3.1 **Translating by hand: two motivating examples**

Databases and Prolog use the the closed world assumption where any object which is not known to be an instance of concept *C* is treated as an instance of ¬*C*. In contrast with this, the open world assumption (OWA) is used in classical logic reasoning, and thus in DL reasoning as well. When reasoning under OWA, one is interested in obtaining statements which hold in all models of the knowledge base, i.e. those entailed by the knowledge base.

A famous DL example of open world reasoning, presented in e.g. [4], is about the family of Oedipus and Iocaste. Let us consider the description logic knowledge base presented in Figure 2.1.

```
∃hasChild.(Patricide ∨ ∃hasChild.¬Patricide) ⊑ Ans
hasChild(Iocaste,Oedipus). hasChild(Iocaste,Polyneikes).
hasChild(Oedipus,Polyneikes). hasChild(Polyneikes,Thersandros).
Patricide(Oedipus). ¬Patricide(Thersandros).
```

Figure 2.1: The Iocaste knowledge base

The only axiom of the TBox is shown in line 1, while the content of the ABox is given in lines 3–5. The TBox axiom expresses that somebody is considered to be an answer if she has a patricide child (a child who killed her father), who in turn has a non-patricide child. The ABox axioms describe the hasChild relation between certain individuals and also express the facts that Oedipus is known to be patricide, while Thersandros is known to be non-patricide. Our task is to solve the instance-check problem Ans(Iocaste), i.e. to decide if Iocaste is deduced to be an answer wrt. the given knowledge base.

Note that Iocaste can be classified as an answer, in spite of the fact that one cannot name the child of Iocaste who has the desired property. That is, solving this specific instance check problem requires case analysis: the child in question is either Polyneikes or Oedipus, depending on Polyneikes being a patricide or not. Also note that the trivial Prolog translation of the DL knowledge base in Figure 2.1, shown below, is not appropriate, as the goal :- Ans(i) fails.

```
Ans(A) :- hasChild(A, B), Patricide(B), hasChild(B, C), not_Patricide(C).
Patricide(o). not_Patricide(t).
hasChild(i, o). hasChild(i, p). hasChild(o, p). hasChild(p, t).
```
Here, to follow the standard DL notation, predicate names corresponding to concepts start with capitals, while role names are written in lower case. For the sake of compactness we omit the apostrophes around Prolog predicate names starting with capitals and we use the abbreviations i, o, p, t for instance names.

Note that using the \+ operator (negation as failure) would not solve the problem: when not_Patricide(C) is replaced by \+ Patricide(C), every instance not known to be patricide would be considered non-patricide, which is not correct. For example, consider the ABox containing axioms \{hasChild(i1, i2), hasChild(i2, i3), Patricide(i2)\}. Here i1 cannot be deduced to be an answer, but the Prolog program using negation as failure would do exactly this.

One can easily see that there is an infinite number of ABox patterns which allow an individual to be proved to belong to concept \textit{Ans} \cite{104}. These patterns are summarised in Figure 2.2. Here the nodes of the pattern graph stand for individuals, while the edges represent the hasChild role instances. Furthermore, \(P\) and \(\neg P\) stand for \textit{Patricide} and \textit{not Patricide}, respectively. Note that case \(n = 2\) corresponds to the ABox given in Figure 2.1.

![Pattern Graph](image)

Figure 2.2: Iocaste ABox patterns.

A Prolog program, written by hand, solving the Iocaste problem is shown in Figure 2.3. We have shown in \cite{104} that this program is a sound and complete translation of the Iocaste problem and it captures exactly the patterns shown in Figure 2.2. To see this, notice that \texttt{dPatricide(Z, X)} describes patterns of form shown in Figure 2.4. The first clause of \texttt{dPatricide(Z, X)} (line 3) corresponds to the degenerate pattern for the \(n = 1\) case. The second clause (line 4) specifies that when the pattern corresponding to \texttt{dPatricide(Y, X)} is extended by two new hasChild edges between \((X, Y)\) and \((Y, Z)\), then a pattern corresponding to \texttt{dPatricide(Z, X)} is obtained.

\begin{verbatim}
1 Ans(X) :- hasChild(X,Y), hasChild(Y,Z), not_Patricide(Z), dPatricide(Y,X).
2 dPatricide(Z, _) :- Patricide(Z).
3 dPatricide(Z, X) :- hasChild(X, Y), hasChild(Y, Z), dPatricide(Y, X).
4 Patricide(o). not_Patricide(t).
5 hasChild(i, o). hasChild(i, p). hasChild(o, p). hasChild(p, t).
\end{verbatim}

Figure 2.3: A correct translation of the Iocaste knowledge base.

Note that the program in Figure 2.3 may not terminate if the hasChild relations form a directed cycle in the ABox. If this cannot be excluded, then termination can be ensured, for example, by tabling \cite{138}, or loop elimination (see Section 2.3.3).
n = 1

\[ P \rightarrow X \]

\[ P \rightarrow Z \]

\[ Y \rightarrow Z \]

Figure 2.4: The pattern captured by dPatricide/2.

Unlike the Iocaste problem, we do not always need to use case analysis and therefore we can generate simpler programs. For example, let us consider the DL knowledge base presented in Figure 2.5. Here we consider someone happy if she has a child who in turn has both a clever child and a pretty child (line 1).

\[
\exists \text{hasChild.}(\exists \text{hasChild.Clever} \cap \exists \text{hasChild.Pretty}) \subseteq \text{Happy}
\]

\[
\text{Clever(lisa). Pretty(lisa). hasChild(kate,bob). hasChild(bob,lisa)}.
\]

Figure 2.5: The Happy knowledge base

Using the ABox given in line 3, we can conclude that kate is happy. In this case, there is a straightforward Prolog translation, as shown below.

\[
\text{Happy(A)} : - \text{hasChild}(A, B), \text{hasChild}(B, C), \text{hasChild}(B, D), \text{Clever}(C), \text{Pretty}(D).
\]

\[
\text{Clever(lisa). Pretty(lisa). hasChild(kate, bob). hasChild(bob, lisa)}.
\]

One of the aims in the DLog project is to create a framework where simple problems result in simple Prolog programs. As we show later in Sections 2.3.6 and 2.4, we can actually generate programs for the Iocaste and Happy problems that either match or are very close to the handmade programs presented here.

### 2.3.2 Building DL clauses from a $\mathcal{SHIQ}$ knowledge base

In this section we deal with the first step of the $\mathcal{SHIQ}$ to Prolog transformation: converting a $\mathcal{SHIQ}$ knowledge base $KB$ to a set of first order clauses of a specific form. First we give an exact specification of this format, and then we define a generalisation which will be used throughout this section.

To transform DL axioms into FOL equivalents we use the techniques described in [99] and [142], which we briefly outlined in Sections 2.2.2 and 2.2.3. In the present chapter we only make use of the fact that the output of these transformations takes a specific form: for an arbitrary $\mathcal{SHIQ}$ knowledge base $KB$, the resulting first-order clauses, denoted by $DL(KB)$, are of the form listed in Figure 2.6.

In this figure clauses (1)–(4) correspond to $KB_T$, while clauses (5)–(6) are simple ABox clauses. $P(x)$ denotes a nonempty disjunction of possibly negated unary literals: $(\lnot)P_1(x) \lor \ldots \lor (\lnot)P_n(x)$. Clause (4) requires further explanation. Its negative binary literals contain all the variables of the clause. Furthermore, if we build a graph from the binary literals by converting $R(x,y)$ to an edge $x \rightarrow y$, this graph will always be a tree. In addition to the binary literals the clause contains unary literals and a possibly empty set of variable equalities.

Note that, as opposed to [99], all clauses containing function symbols are eliminated during saturation: the resulting clauses can be resolved further only with ABox clauses. This forms the basis of a pure two phase reasoning framework, where every possible ABox-independent reasoning is done before accessing the ABox itself, allowing us to store the content of the ABox in an external database.
In the remaining part of this chapter we focus on how to transform clauses of form shown in Figure 2.6 into efficient Prolog code. However, we note that for the general transformation we use only certain properties of the clauses. These properties are satisfied by a subset of first order clauses that is, in fact, larger than the set of clauses that can be generated from a SHIQ KB. These properties are summarised in the following definition.

**Definition 1.** A clause $C$ is said to be a DL clause if it satisfies the following properties.

1. $C$ is function free, i.e. there is no literal in $C$ that contains function symbols.
2. $C$ either contains a binary literal or it contains at most one variable.
3. If there is a binary literal in $C$ then each variable in $C$ occurs in at least one binary literal.
4. If $C$ contains a positive binary literal $B$, then all remaining literals $C' = C \setminus \{B\}$ are negative binary literals, and the set of variables of $C'$ and $B$ is the same.

Now we formulate the following proposition (simply proved by checking each of the clauses in Figure 2.6).

**Proposition 1.** For a given SHIQ knowledge base $KB$, every clause $C \in DL(KB)$ is a DL clause.

Note that the properties in Definition 1 are a necessary but not a sufficient condition for being a clause of form shown in Figure 2.6, i.e. these properties may also hold for a clause that cannot be derived from any SHIQ knowledge base. An example for such a clause is the following:

$$P(x) \lor \neg R(x,x)$$

In the rest of this section when describing the general SHIQ to Prolog transformation, we deal with clauses satisfying the conditions described in Definition 1. Note however that in Section 2.4, describing the optimisations of this transformation process we will restrict our attention to SHIQ KBs. As a consequence of this, the results of this section can be used for DL knowledge bases that are more expressive than SHIQ. This includes the use of certain role constructors, such as union and intersection. Furthermore, parts of the knowledge base can be formulated directly as first order clauses satisfying properties (p1)–(p4), such as clause (2.1) above.

Now we turn our attention towards certain types of atomic predicates. The following definition introduces the notion of a top predicate, which holds for every possible $x$.

**Definition 2.** Let $S$ be a set of DL clauses and let $p$ be an atomic predicate name that appears somewhere in $S$. Predicate $p$ is said to be a top predicate if $S$ entails $p(x)$.

We would like to eliminate every top predicate from a given set of DL clauses. This is because for a top predicate a FOL theorem prover returns without instantiating $x$. In contrast with this, a DL reasoner would be expected to enumerate all individuals in an ABox, as the answers to an instance retrieval query concerning a concept corresponding to a top predicate. The elimination process results in a reduced set of DL clauses as defined below.
**Definition 3.** Let \( S \) be a set of DL clauses. We modify \( S \) in the following way. (1) We remove all literals in \( S \) which contain a negated top predicate. (2) We remove every clause \( C \) from \( S \) where \( C \) contains a positive literal with a top predicate. The remaining set of clauses is called the *reduced* version of \( S \).

The following proposition shows that this reduction step preserves all information except for the top predicates.

**Proposition 2.** Let \( S \) be a set of DL clauses. Let us extend the *reduced form* of \( S \) with clauses of form \( p(x) \), for each top predicate \( p \) in \( S \). This extended set of clauses is equivalent to \( S \).

**Proof.** Easily follows from the fact that the modifications in the reduction process are sound. \( \square \)

In the following, we will restrict our attention to sets of DL clauses that are in reduced form.

### 2.3.3 Specialising PTTP for DL clauses

As the clauses of a \( SHIQ \) knowledge base \( KB \) are normal first-order clauses we can apply the PTTP technology (cf. Section 2.2.4) directly on these. This means that we first have to generate the contrapositives of \( DL(KB) \), which also involves the introduction of new predicate names for negated literals. This process, in a slightly generalised form, is encapsulated in the following definition.

**Definition 4.** By the *DL program* corresponding to a set of DL clauses \( S \), denoted by \( PDL(S) \), we mean the set of Horn clauses that are the contrapositives of \( S \), i.e. \( PDL(S) = \{ C | C \) is a contrapositive of \( C_0 \) and \( C_0 \in S \} \).

When this operation is applied to \( DL(KB) \), i.e. a set of DL clauses derived from the \( SHIQ \) knowledge base \( KB \), we obtain the set of Horn clauses \( PDL(DL(KB)) \). By overloading the name \( PDL \), this will be abbreviated as \( PDL(KB) \).

Horn clauses are usually grouped into predicates, according to the functor of the clause head. Thus a DL program can be also viewed as a set of *DL predicates*, each of which consists of all clauses of the DL program which have a given head functor. In the rest of the chapter it is the context which determines which view of DL programs is used.

As an example, we show the four DL predicates of the Iocaste knowledge base in Figure 2.7. Notice, for example, that the first clause of the *Patricide/1* predicate comes from the ABox, while the second comes from the TBox. We also show the five DL predicates of the Happy KB in Figure 2.8.

```prolog
Ans(A) :- hasChild(A, B), hasChild(B, C), Patricide(B), not_Patricide(C).
Patricide(o).
Patricide(A) :- hasChild(B, A), hasChild(C, B), Patricide(B), not_Ans(C).
not_Patricide(t).
not_Patricide(A) :- hasChild(A, B), hasChild(C, A), not_Ans(C), not_Patricide(B).
hasChild(i, o). hasChild(i, p). hasChild(o, p). hasChild(p, t).
```

**Figure 2.7:** DL predicates of the Iocaste problem

Having defined the notion of DL programs let us now discuss how such a program can be queried. Traditional instance retrieval queries include positive and negated atomic concepts and positive binary roles. Example for such queries are \( \text{not}_\text{Patricide}(A) \) or \( \text{hasChild}(A, B) \). The former is supposed to enumerate all possible individuals known to be non-Patricide. The latter should enumerate all possible pairs of individuals between whom the hasChild relation holds. In this chapter we support *conjunctive queries* which are a generalisation of instance retrieval problems: a conjunctive query is a conjunction of
Figure 2.8: DL predicates of the Happy problem

the above mentioned instance retrieval constructs [44]. An example conjunctive query is (Patricide(X), hasChild(X, Y), not_Patricide(Y)).

For DL predicates the PTTP technology provides a sound and complete reasoning framework. Let us now examine how the PTTP techniques can be simplified in the special case of DL clauses. First, let us notice that occurs check is not necessary, as DL clauses are function free. Next, in the case of conjunctive queries, we claim that (1) ancestor resolution is not needed for roles and (2) contrapositives with negated binary literals in the head can be eliminated. To prove these claims let us first observe the following proposition.

**Proposition 3.** In a DL program, a negated binary predicate can only be invoked within a negated binary predicate.

**Proof.** Let C be a clause in the DL program, such that the body of C contains a negated binary goal G. Accordingly, C is the contrapositive of a DL clause where the binary literal corresponding to G is a positive literal. However, because of property (p4) in Definition 1, we know that this DL clause cannot contain any more positive binary literals and, moreover, it can only contain negative binary literals. Thus, the head of C must correspond to a negative binary literal. □

We have actually proved that a negated binary predicate can only appear in the ancestor list if another negated binary predicate precedes it. This directly leads to the following proposition.

**Proposition 4.** Ancestor resolution is not required for binary predicates to answer conjunctive queries wrt. a DL program S.

**Proof.** From Proposition 3 we know that a negated binary predicate can only be invoked within a negated binary predicate. As a conjunctive query does not allow goals invoking such predicates, the ancestor list built during the execution of the query can only contain unary and positive binary predicates. As negative binary predicates are never called and never occur in the ancestor list, ancestor resolution for binary predicates can never be applied. □

Another useful conclusion is

**Proposition 5.** Removing contrapositives with negated binary literals in the head from a DL program does not affect the execution of conjunctive queries.

**Proof.** This is a direct conclusion of Proposition 3 and the fact that a conjunctive query cannot contain negated binary predicates. □

Note that when the clauses with negated binary literals in the head are removed, no negative binary literals will remain in the bodies (as the latter only appear in clauses with negative binary heads, cf. the proof of Proposition 3). Thus, in the following, the term binary predicate will refer to positive binary predicates.
Let us now introduce another simplification of the PTTP approach for DL clauses: instead of iterative deepening we can use the normal Prolog depth-first search if we apply a simple loop elimination technique. This feature, which results in pruning certain branches of the Prolog execution tree, already appeared in PTTP, as an optimisation [125]. However, in the context of DL predicates, as opposed to arbitrary first order logic clauses, loop elimination can itself ensure termination, as discussed below. First, let us observe

**Proposition 6** (binary instantiation). Let \( S \) be the set of DL predicates and \( B \) be a binary goal. If the Prolog execution of \( B \) w.r.t. \( S \) terminates with success, it instantiates both of its arguments.

**Proof.** Let us indirectly assume that there is a binary goal \( B(X, Y) \) which terminates, but one of its arguments, let us say \( X \), remains uninstantiated. Since \( B(X, Y) \) terminates, there is a finite Prolog proof tree \( T \) for it. Let us consider the nodes in \( T \) containing a binary goal with \( X \) as one of its arguments. As \( T \) is finite, there exists a “lowest” of these, i.e. a node with no occurrences of \( X \) in binary goals below it. However, this contradicts property (p4) in Definition 1. \( \square \)

We now continue with several further definitions. First we define the notion of binary-first body ordering.

**Definition 5.** The body of a Prolog clause \( C \) is ordered according to the binary-first rule if a binary goal \( B \) with variable \( V \) comes before all the unary goals containing \( V \).

Using this definition of binary-first rule we make

**Proposition 7** (non-groundness of unary predicates). Let \( S \) be a set of DL predicates and let us use the binary-first rule during the Prolog execution. If a unary predicate is invoked with a variable argument then its parent goal is a unary predicate with the same variable.

**Proof.** According to Definition 1, a unary goal \( G \) with a variable argument \( V \) can only be invoked from within a clause \( C \) of a unary predicate. If there are binary goals within the body of \( C \), then \( G \) is always preceded by a binary goal containing \( V \) according to (p3) in Definition 1 and the binary-first rule. Because of Proposition 6, however, we know that this variable is already instantiated by the time \( G \) is invoked.

This means the body of \( C \) contains only unary goals and so, according to (p2) in Definition 1, \( G \) has the same variable that appears in the head of \( C \). \( \square \)

We continue with the definition of a goal being subject to loop elimination.

**Definition 6.** A goal \( G \) encountered in the context of an ancestor list \( L \) is subject to loop elimination if \( G \) occurs in \( L \): more precisely, if \( L \) contains an element \( G' \) for which \( G == G' \) holds. Here \( == \) denotes the standard Prolog predicate which succeeds if its operands are identical.

Now we define loop elimination as an extension to the Prolog execution mechanism, assuming that the list of ancestor goals is maintained, and available for inspection during execution.

**Definition 7.** Let \( P \) be a Prolog program. By loop elimination we mean the Prolog execution of \( P \) which is extended in the following way: we stop the given execution branch with a failure whenever we encounter a goal \( G \) that is subject to loop elimination.

Using the notion of loop elimination we can formulate

**Proposition 8** (termination of DL predicates). Let \( S \) be the set of DL predicates. Using loop elimination and the binary-first rule the execution of any goal in \( S \) always terminates.

**Proof.** Let us indirectly assume that there exists a goal \( G \) the execution of which does not terminate. Because of loop elimination this can only happen if we can build an ancestor list with infinitely many distinct goals. Since the number of predicate and constant names is finite this means that the ancestor list contains an infinite number of distinct variables.

Let us examine when a new variable can appear in the ancestor list. This cannot happen in a unary predicate invocation, because of Proposition 7. This cannot happen either when invoking a binary goal from a binary predicate because of (p4) in Definition 1. Thus a new variable can only be introduced when a binary goal is invoked in an unary predicate.
This means we have an infinite sequence of nested goals where a unary predicate directly calls a binary predicate which in turn, possibly indirectly, calls a unary predicate, etc. In this sequence of nested unary predicates all but the first have to be ground because of Proposition 7.

However, this means that there exists an infinite sequence of distinct ground unary goals which is a contradiction because the number of predicate and constant names is finite.

Now that we have proved that loop elimination and the binary-first rule guarantee termination, let us consider the issue whether loop elimination is complete, i.e. any solution that can be obtained by PTTP can also be obtained in the presence of loop elimination.

Note that for normal Prolog execution, loop elimination is obviously complete. That is, given an arbitrary proof tree of a goal $P$ where goal $G_1$ appears in the subtree of an identical goal $G_2$ we can always create a new proof tree of $P$ where we replace the proof of $G_1$ by the proof of $G_2$. Continuing this process we can obtain a proof tree of $P$ that does not contain any goal subject to loop elimination.

However, PTTP extends the normal Prolog execution by applying ancestor resolution for goals. This means that successful execution of a goal $G$ may depend on the location of $G$ within a proof tree (as this determines the ancestors of $G$). Thus it is important to ensure the completeness of loop elimination in the presence of ancestor resolution. The following is a reformulation of the proposition that appeared in [126].

**Proposition 9** (completeness of loop elimination). Let $T$ be a proof tree of a goal $G$ corresponding to a PTTP execution, which contains a goal subject to loop elimination. It is possible to create another proof tree of goal $G$ which contains no goals subject to loop elimination.

We now turn our attention towards some issues related to ancestor goals. Both loop elimination and ancestor resolution relies on the relationship of a predicate invocation with one of its ancestors. Let us now examine what ancestor–descendant pairs are possible for unary predicates. In general, we have the following five cases, where variables $X$ and $Y$ are distinct, but predicate names $q$ and $p$, as well as constants $i$ and $j$ can be the same.

(c1) for proving $p(i)$ we need to prove $q(j)$
(c2) for proving $p(i)$ we need to prove $q(X)$
(c3) for proving $p(X)$ we need to prove $q(i)$
(c4) for proving $p(X)$ we need to prove $q(X)$
(c5) for proving $p(X)$ we need to prove $q(Y)$

The following proposition states that, in the case of DL predicates, some of these cases cannot happen.

**Proposition 10.** Let $S$ be the set of DL predicates. When using the binary-first rule, cases (c2) and (c5) cannot occur during Prolog execution.

*Proof.* This claim is a direct consequence of Proposition 7.

Let us now examine what special properties hold in ancestor resolution of DL predicates. For this, let us examine cases (c1), (c3) and (c4) for $q = \text{not}_p$, i.e. the case relevant for ancestor resolution. In the case of (c1), ancestor resolution succeeds if $i = j$, and fails otherwise. Note that this ancestor resolution step can succeed only once. This is because loop elimination ensures that the ancestor list cannot contain $p(i)$ more than once. In (c3), variable $X$ in the ancestor is instantiated to $i$. Case (c4) succeeds with no substitution. Similarly to (c1), both (c3) and (c4) can succeed only once as $p(X)$ cannot occur in the ancestor list more than once. Thus we have actually proved the following proposition.

**Proposition 11** (deterministic ancestor resolution). If loop elimination is applied for DL predicates, exactly one ancestor can be applicable in a successful ancestor resolution step, i.e. ancestor resolution is deterministic.

To conclude this section, in Figure 2.9 we summarise the principles we use in the execution of DL predicates and compare these to their counterparts in PTTP. We also formulate the main result of this section as the following theorem.
DLog uses normal Prolog unification rather than occurs check

DLog uses loop elimination instead of iterative deepening

DLog eliminates contrapositives with negated binary literals in the head

DLog does not apply ancestor resolution for roles

DLog uses deterministic ancestor resolution

Figure 2.9: Principles of DLog execution in contrast with PTTP.

**Theorem 1** (soundness and completeness of the DLog execution). Let $S$ be a set of DL clauses in reduced form and $Q$ a conjunctive query. Let $P$ be a set of Prolog clauses obtained from $PDL(S)$ by removing clauses with negated binaries in the head, and by ordering clause bodies according to the binary-first rule. Let us extend a standard Prolog engine with (1) loop elimination and (2) deterministic ancestor resolution for unary predicates only. If the extended Prolog engine is invoked with the program $P$ and goal $Q$, it will terminate and enumerate those and only those constant instantiations of the variables of $Q$ for which $Q$ is entailed by $S$.

**Proof.** This is a direct consequence of the fact that PTTP is a sound and complete FOL theorem proving technique and of Propositions 5, 8, 9 and 11.

**Corollary 1.** Let $KB$ be a $SHIQ$ knowledge base and $Q$ a conjunctive query in which no concept equivalent to $\top$ occurs. In this case the technique of Theorem 1, applied to the reduced form of DL($KB$) and $Q$ provides finite, sound and complete query execution.

### Interpreting DL predicates

In Figure 2.10 we show a complete $SHIQ$ interpreter that is able to execute DL predicates stored as normal dynamic predicates in Prolog.

```prolog
interp(true, _) :- !.
interp([Goal1, Goal2], AncList) :- !,
    interp(Goal1, AncList),
    interp(Goal2, AncList).
interp(Goal, AncList) :-
    \{ member(Goal0, AncList), Goal0 == Goal -> fail % loop elimination
    ; neg(Goal, NegGoal), memberchk(NegGoal, AncList) % ancestor resolut.
    ; NewAncList = [Goal|AncList],
    clause(Goal, Body),
    interp(Body, NewAncList)
    \}.
```

Figure 2.10: A full $SHIQ$ interpreter

The interpreter is invoked through predicate $interp/2$ with a goal in the first, and an empty ancestor list in the second argument. The interpreter provides loop elimination (line 7) and deterministic (cf. the use of $memberchk/2$) ancestor resolution (line 9). The new ancestor list is built in line 11. Term $NegGoal$ in line 9 is the negated version of $Goal$ according to the following definition.
Definition 8. Let $K$ be a name. The negated name of $K$, denoted by $\neg(K)$, is defined as follows.

\[
\neg(K) = \begin{cases} 
S & \text{if the form } K = \text{not}_S \\
\text{not}_K & \text{otherwise}
\end{cases}
\]

Let $H$ be an arbitrary Prolog term with name $N$ and arguments $A_1, \ldots, A_k$. By $\neg(H)$ we denote the term $\neg(N)(A_1, \ldots, A_k)$, i.e. a variant of $H$ which differs from $H$ only in that the name is negated.

For example, $\neg(p(i))$ is $\text{not}_p(i)$.

Now, as an example of using the interpreter, when the DL predicates of the Iocaste problem, as shown in Figure 2.7, are loaded, we can run the Iocaste query in the following way:

```
| ?- interp('Ans'(X), []). 
X = i ; 
no
```

According to Theorem 1, the interpreter is a sound and complete theorem prover for DL programs and composite queries, assuming that a binary-first body ordering is used.

### 2.3.5 Compiling DL predicates

Although the interpreted solution is pretty straightforward, for performance reasons we also consider generating Prolog code that requires no special interpreter. The idea is to include loop elimination and ancestor resolution in the DL predicates themselves (below we always assume that their bodies are ordered according to the binary-first rule).

In contrast with the interpreter, the compilation treats TBox and ABox clauses separately. Therefore we now distinguish between TBox and ABox DL predicates: $PDL(KB_T)$ and $PDL(KB_A)$, respectively. For example, in Figure 2.7, clauses in lines 3, 6 and 10 form the ABox DL predicates, while the rest contain the TBox DL predicates.

To precisely describe the transformation process we now define the notion of signature.

**Definition 9.** Let $KB$ be an SHIQ knowledge base. The signature of the TBox (ABox) DL predicates of $KB$ is the set of functors of form $C/1$ and $R/2$ where $C$ is a concept name and $R$ is a role name that appears anywhere in $PDL(KB_T)$ ($PDL(KB_A)$).

For example, the signature of the ABox part of the Iocaste DL predicates shown in Figure 2.7 is the set \{Patricide/1, not_Patricide/1, hasChild/2\}, while the signature of the TBox part is \{Ans/1, not_Ans/1, Patricide/1, not_Patricide/1, hasChild/2\}.

We now define two transformations: expansion and ancestorisation.

**Definition 10.** Let $T$ be an arbitrary Prolog term with name $N$ and arguments $A_1, \ldots, A_k$. Let $Z$ be another Prolog term. The expanded version of $T$ wrt. $Z$, denoted by $\text{Expd}(T, Z)$ is defined as the term $N(A_1, \ldots, A_k, Z)$.

**Definition 11.** For an arbitrary Prolog clause $C$ with head $H$ and body $B_1, \ldots, B_n$ the ancestorised form of $C$, $\Omega(C)$, is a Prolog clause defined as follows. The head of $\Omega(C)$ is $\text{Expd}(H, AL)$, where $AL$ is a newly introduced variable. The body of $\Omega(C)$ is $E_0, E_1, \ldots, E_n$. $E_0$ is the goal $\text{AncL} = [H | AL]$ and $E_i = \text{Expd}(B_i, \text{AncL})$, for $i > 0$.

As an example, the ancestorised form of the Iocaste clause shown in lines 1–2 in Figure 2.7 is the following:

```
\begin{verbatim}
Ans(A, B) :- C = [Ans(A) | B], hasChild(D, E, C), hasChild(A, D, C), Patricide(D, C), not_Patricide(E, C).
\end{verbatim}
```

Here $B$ denotes the old, while $C$ denotes the updated ancestor list. We can now define the notion of the compiled form of a TBox DL predicate.
**Definition 12.** Let \( P \) be a TBox DL predicate with functor \( N/A \) and its clauses \( C_1, \ldots, C_n, n \geq 0 \). Let \( H \) denote a term with name \( N \) and arity \( A \), where every argument is a distinct variable. By \( \Delta(P) \) we denote a sequence of clauses \( F_1, \ldots, F_{n+3} \), called the compiled version of \( P \). Here, clauses \( F_1, F_2 \) and \( F_3 \) are the following:

- \( F_1: \text{Expd}(H, \text{AL}) :- \text{member}(G, \text{AL}), \ G==H, \ !, \ \text{fail.} \) (cf. line 7 in Figure 2.10)
- \( F_2: \text{Expd}(H, \text{AL}) :- \text{memberchk}(\neg(H), \text{AL}). \) (cf. line 9 in Figure 2.10)
- \( F_3: \text{Expd}(H, \text{AL}) :- H. \)

Finally, \( F_{3+i} = \Omega(C_i), i > 0 \), i.e. the remaining clauses of \( \Delta(P) \) are the ancestorised versions of the clauses \( C_1, \ldots, C_n, n \geq 0 \).

This definition says that the compiled version of a set of clauses is the ancestorised version of these clauses extended with three new clauses. These new clauses are responsible for loop elimination, ancestor resolution and for accessing the content of the ABox. We note that for roles we do not have to generate \( F_2 \) clauses (cf. Figure 2.9), and if \( N/A \) does not appear in the ABox signature, we can also omit clauses of type \( F_3 \).

We also note that \( \Omega \) can be applied to an empty set of clauses: the compiled version of the empty set is the sequence of clauses \( F_1, F_2 \) and \( F_3 \). Now if we omit \( F_3 \) based on the ABox signature we have a special case: this predicate can only succeed by ancestor resolution, i.e. using clause \( F_2 \). Note that such predicates do not need loop elimination (i.e. we can omit clause \( F_1 \)) as they do not invoke any other predicate. This type of predicates is covered by

**Definition 13.** A predicate with only a single clause of type \( F_2 \) is called orphan and any invocation of such a predicate is called orphan goal or orphan call.

We can also give an alternative definition of an orphan goal: a goal \( G \) is orphan in a DL program if it appears in a clause body, but it does not appear in the head of any of the clauses.

Let us denote by \( \Delta'(P) \) the compiled form \( \Delta(P) \) possibly modified by the transformations described above. Now we are finally ready to define the compiled form of a DL program.

**Definition 14.** Let \( KB \) be an S\&H\&I\&Q knowledge base, and let \( S = \{N_1/A_1, \ldots, N_k/A_k\} \) be the signature of \( PDL(KB_T) \). The compiled form of the DL program of \( KB \), i.e. \( PDL(KB) \), is the set \( \{C_1, \ldots, C_k\} \cup PDL(KB_T) \), where \( C_i = \Delta'(Z_i) \) and \( Z_i = \{C \in PDL(KB_T) \mid \text{head of } C \text{ has functor } N_i/A_i\} \).

Accordingly, we obtain the compiled form of a set of DL predicates \( P \) by compiling the predicates belonging to every functor appearing in the TBox DL predicates of \( P \), together with the unmodified ABox DL predicates. In the following we give detailed examples of the compilation process.

### 2.3.6 Compilation examples

As discussed above the general idea is that we add appropriate pieces of Prolog code to the DL clauses to handle ancestor resolution and loop elimination. We demonstrate this technique by presenting the complete Prolog translation of the Locaste DL predicates presented in Figure 2.7. The result is shown in Figure 2.11.

In this program, most of the predicates have an extra argument: as explained earlier these are used to pass the ancestor list from call to call. For example, in line 10 we invoke the goal \text{Patricide}(D, C) \), where \( C \) already contains the new ancestor list constructed in line 9.

In general, the content of the ABox can be either described as Prolog facts, as we did in lines 20–21, or can be stored externally in some database. In the latter case we have to only provide “stubs” to access the content of the ABox. Namely, we should provide three clauses, one for \text{Patricide}/1, one for \text{not_Patricide}/1 and another one for \text{hasChild}/2. These clauses should instantiate their head variables by properly querying the underlying database. This way, we are able to handle really large ABoxes that physically would not fit into memory. In the following, for the sake of simplicity, we describe the content of the ABox as Prolog facts in the generated programs. Predicates consisting of facts only are called atomic.

Note that as the Locaste and Happy examples do not contain role axioms, we only have atomic binary predicates. Thus in contrast with the general compilation scheme presented in Section 2.3.5, these roles do
not have an extra argument: any role invocation appears unmodified in the code (e.g. hasChild(D, E) in line 3).

In Figure 2.11, the first two clauses of the predicates are responsible for loop elimination and ancestor resolution. The first clauses, i.e. clauses in lines 1, 6 and 12, check whether the ancestor list contains the goal in question. If so, this means we are just about to fall into an infinite loop. Therefore, as already explained in Section 2.3.2, we can cut the rest of the clauses and simply fail.

Clauses in lines 2, 7 and 13 are used to check whether the ancestor list contains the negation of the goal in question. If this is the case, then an ancestor resolution step happens with a possible substitution of variable A. As explained earlier we leave a choice point here so that the remaining clauses of the given predicate can be executed if, for example, the branch using the ancestor resolution fails.

Finally, line 18 in Figure 2.11 demonstrates the translation of an orphan predicate.

Applying all the techniques we get a complete Prolog translation: if we load the program in Figure 2.11 and enumerate the instances of the concept Ans...

| ?- ‘Ans’(X, []). |

...we get the X = i instantiation, as an answer.

Let us now compare the handmade translation of the Iocaste problem in Figure 2.3 with the machine translation shown above. Here, Ans(X) in the former corresponds to Ans(X, []) in the latter, and similarly dPatricide(X, Y) corresponds to Patricide(X, [...Ans(Y)...]). Notice that in the handmade version the top individual of the Iocaste pattern (i.e. Iocaste herself) is passed through an extra argument to dPatricide/2. The same effect is achieved in the machine translation by placing Ans(Y) on the ancestor list and retrieving it later using ancestor resolution.

A further difference is that in the handmade variant predicate not_Patricide/2 does not appear. This is because, in this specific case it is not needed for capturing the patterns in Figure 2.2.

Also note that the predicates in the machine translation have more clauses than in the handmade version. Some of these, such as the clause responsible for loop elimination in Ans/2 and the one responsible for ancestor resolution in Patricide/2, are superfluous, and will actually be removed by optimisations
presented in Section 2.4. However, the clause ensuring loop elimination in Patricide/2 has to stay, as without it termination cannot be assured in the presence of cyclic hasChild relations.

To conclude the demonstration of the general approach we also show the translation of the Happy knowledge base in Figure 2.12 according to the generic compilation scheme. Note that a new ancestor list is built in lines 8 and 13. As a trivial simplification, we do not build such a new ancestor list if it is not passed to any of the goals in the body of a clause. This happens if the clause invokes atomic predicates only (lines 3–4) to which we do not pass the ancestor list at all.

```
Happy(A, B) :- member(C, B), C == Happy(A), !, fail.
Happy(A, B) :- memberchk(not_Happy(A), B).
Happy(A, _) :- hasChild(A, B), hasChild(B, C), hasChild(B, D), Clever(C), Pretty(D).

not_Clever(A, B) :- member(C, B), C == not_Clever(A), !, fail.
not_Clever(A, B) :- memberchk(Clever(A), B).
not_Clever(A, B) :- F = [not_Clever(A)|B], hasChild(C, A), hasChild(D, C), hasChild(C, E), Pretty(E), not_Happy(D, F).

not_Pretty(A, B) :- member(C, B), C == not_Pretty(A), !, fail.
not_Pretty(A, B) :- memberchk(Pretty(A), B).
not_Pretty(A, B) :- F = [not_Pretty(A)|B], hasChild(C, A), hasChild(D, C), hasChild(C, E), Clever(E), not_Happy(D, F).

not_Happy(A, B) :- memberchk(Happy(A), B).
Clever(lisa). Pretty(lisa). hasChild(kate, bob). hasChild(bob, lisa).
```

Figure 2.12: A naive Prolog translation solving the Happy problem

2.4 Optimising DL compilation

Although the translation principles presented in the previous section are complete and result in programs that can already be executed in a standard Prolog environment they are not very efficient. In this section we describe a series of optimisations that result in a much more efficient Prolog translation. We note that although in principle, these optimisations could also be built in to the interpreter itself, here we deal with the compiled form only.

In Section 2.3 we introduced the general interpretation and compilation schemes for so called DL clauses, allowing more general knowledge bases than $SHIQ$. In contrast with that, the optimisations presented here are designed only for DL clauses derived from a $SHIQ$ KB as shown in Figure 2.6. Although we believe that several optimisation schemes are also applicable to DL clauses in general (as defined in Definition 1), making such generalisations forms part of our future work.

2.4.1 Principles of optimisation

The complete optimisation process is summarised in Figure 2.13. At the very first step we do filtering: we remove those clauses that need not to be included in the final program as they are not used in the execution (see Section 2.4.2).

Next, we categorise the remaining clauses (see Section 2.4.3). This provides us with valuable information, as from specific groups of predicates we can generate more efficient Prolog code. The first two optimisations are global, in the sense that e.g. the condition of removing a clause in the filtering phase may depend on other parts of the knowledge base.
Using the categorisation information we then apply several optimisations. Most of these optimisations are local in a sense that they are applied in a local context, e.g. on a single predicate. The optimisations are independent from each other: any combination of these can be used when generating the final Prolog program (cf. arrows in Figure 2.13). These optimisations are summarised below. Note that some further, lower level optimisations are introduced in Section 2.5.2.

(o1) ordering of goals in clause bodies (Section 2.4.4)
(o2) support for multiple argument indexing (Section 2.4.5)
(o3) efficient ground goal execution (Section 2.4.6)
(o4) decomposition of clause bodies (Section 2.4.7)
(o5) projection for eliminating multiple answers (Section 2.4.8)
(o6) efficient translation of roles and inverses (Section 2.4.9)

Note that all the above optimisations except for (o2) and (o6) concern unary predicates. Therefore in the subsections corresponding to (o1) and (o3)–(o5) we implicitly assume that all clauses belong to unary predicates.

Before we go into details we introduce some definitions to be used in the upcoming sections. First we introduce some notions regarding the negation of a predicate. Note that in the following we always refer to a predicate by its functor.

**Definition 15.** Let \( P \) be a predicate with functor \( N/A \). By the negation of \( P \), denoted by \( \text{not}_-P \), we mean a predicate with functor \( M/A \), where \( M = \text{neg}(N) \).

For example, for \( P = \text{not}_{-}\text{Happy}/1 \) in Figure 2.8, \( \text{not}_-P \) denotes predicate \( \text{Happy}/1 \).

Next we introduce the notion of reachability between predicates.

**Definition 16.** A predicate \( P_1 \) **directly calls** predicate \( P_2 \) if \( P_2 \) is invoked in any of the clauses of \( P_1 \). It is possible to **reach** \( P_2 \) from \( P_1 \) if (1) either \( P_1 \) directly calls \( P_2 \) or (2) there exists a predicate \( P_i \) from which it is possible to reach \( P_2 \) and \( P_1 \) directly calls \( P_i \).

Thus, relation **reach** is the transitive closure of the relation **directly calls** between predicates. As an example, let us consider the knowledge base in Figure 2.7. Here \( \text{not}_{-}\text{Ans}/1 \) is reachable from \( \text{Ans}/1 \), although it is not directly called. Note that the definition of reachability can naturally be reformulated for clauses.

**Definition 17.** It is possible to reach predicate \( P_2 \) from clause \( C_1 \) of predicate \( P_1 \) if \( C_1 \) invokes a predicate \( T \) such that \( P_2 \) is reachable from or identical to \( T \).
Note that if we reach a predicate then we reach all of its clauses as well. Thus we can also talk about whether a predicate reaches a clause or a clause reaches another clause. Now we introduce some important properties of the DL predicates.

**Definition 18.** Predicate $P$ is recursive if it is reachable from itself. If $\text{not} \_P$ is reachable from $P$, then $P$ is said to be an ANR (ancestor negative recursion) predicate, i.e. $P$ occurs in the ancestor role in negative recursion [116]. Finally, if predicate $P$ is reachable from $\text{not} \_P$, then $P$ is called a DNR (descendant negative recursion) predicate, i.e. $P$ occurs in the descendant role in negative recursion.

Obviously $P$ is ANR if and only if $\text{not} \_P$ is DNR (using the fact that $\text{not} \_\text{not} \_P$ is $P$).

### 2.4.2 Filtering

In the filtering phase we omit those clauses of the DL predicates that are not required in producing solutions.

**Definition 19.** A clause $C$ in the DL program is called *eliminable* if the body of $C$ cannot succeed under any condition.

In the following we consider two special types of eliminable clauses.

**Definition 20.** Let $C$ be a clause of a predicate $P$ in the DL program. $C$ is said to have the *false-orphan* property if the body of $C$ invokes an orphan predicate $O$ and it is not possible to reach $P$ (and thus $C$) from $\text{not} \_O$.

**Proposition 12.** A clause $C$ having the false-orphan property is eliminable.

*Proof.* Let $C$ be a clause of predicate $P$ having the false-orphan property. By definition, $C$ invokes an orphan predicate $O$ such that it is not possible to reach $P$ from $\text{not} \_O$. Let us denote the functor of predicate $\text{not} \_O$ by $F$. Accordingly, when invoking $C$, the ancestor list cannot contain any goal with functor $F$. However, this means that the invocation of $O$ and thus clause $C$ fails, since we know that any invocation of $O$ can only succeed by ancestor resolution, i.e. by finding a goal in the ancestor list having functor $F$.

Let us now introduce another type of clause which can also be eliminated from the knowledge base.

**Definition 21.** Let $C$ be a clause in the DL program. $C$ is said to have the *two-orphan* property if the body of $C$ invokes orphan predicates $O_1, \ldots, O_n$ from which at least two predicates have different names.

Before stating that such clauses are eliminable, we first make the following useful proposition.

**Proposition 13.** Let $O_1$ and $O_2$ be orphan predicates and let us denote the functors of $\text{not} \_O_1$ and $\text{not} \_O_2$ by $F_1$ and $F_2$, respectively. If the ancestor list contains a goal with functor $F_1$ as well as a goal with functor $F_2$, then either the invocation of $O_1$ or the invocation of $O_2$ has to fail.

*Proof.* Without loss of generality, let us indirectly assume that some invocation of $O_2$ can succeed (i.e. $O_2$ can be reached from $\text{not} \_O_2$) and there is a goal $X$ with functor $F_1$ that comes before goal $Y$ with functor $F_2$ in the ancestor list. Note that $X$ and $Y$ are concrete invocations of predicates $\text{not} \_O_1$ and $\text{not} \_O_2$. This means that it is possible to reach predicate $\text{not} \_O_2$ from predicate $\text{not} \_O_1$, i.e. goal $Y$ is located in the body of some clause $T$ with head $H$, where $T$ is reachable from $\text{not} \_O_1$.

Let us consider now the contrapositive $C$ of clause $T$ where $\text{neg}(Y)$ is in the head, while $\text{neg}(H)$ is in the body. Note that $C$ is a clause of predicate $O_2$, as $\text{neg}(Y)$ denotes a term with the same functor as $O_2$. This, however, contradicts the fact that $O_2$ is an orphan predicate.

Now we only have to show that clause $C$ really exists in the DL program, i.e. it cannot be eliminated because of the false-orphan property. According to Definition 20, Proposition 12 and the fact that $T$ exists, $C$ cannot be eliminated if $C$ is reachable from $T$. This is true, however, as clause $T$, via goal $Y$, calls predicate $\text{not} \_O_2$ and from this predicate $O_2$, and thus clause $C$, is reachable, because of the initial assumption.

Now we can prove the following proposition.
Proposition 14. A clause having the two-orphan property is eliminable.

Proof. Let $C$ be a clause having the two-orphan property. According to the definition, $C$ invokes two orphan predicates $O_1$ and $O_2$ with different names. Since any invocations of $O_1$ and $O_2$ can only succeed by ancestor resolution, goals having the same functors as $not_{O_1}$ and $not_{O_2}$ must be in the ancestor list at the time of executing $C$. According to Proposition 13, however, this means that clause $C$ fails as either the invocation of $O_1$ or $O_2$ has to fail.

We note that a clause containing more than one orphan goal with the same name cannot be eliminated, as all of these goals can succeed by resolving against a single ancestor in the ancestor list.

Another useful conclusion of Proposition 13 is

Proposition 15. Let $O$ be an orphan predicate the invocation of which can succeed by ancestor resolution. In the ancestor list there can never be two goals with the functor of $not_{O}$.

Proof. Our claim is a special case of Proposition 13, where $O_1 = O_2 = O$.

We have actually proved that ancestor resolution for orphan predicates is deterministic. We have proved earlier that ancestor resolution in general is deterministic (cf. Proposition 11), but for this we had to assume that the binary first rule was applied. Note that this assumption is not present in the above Proposition 15.

Definition 22. A DL program is filtered if it does not contain any eliminable clauses.

Note that if we eliminate a clause $C$ which was the only remaining clause of predicate $P$, then any invocation of $P$ in the DL program $C_1, \ldots, C_i$ has actually become an orphan call. This means that now clauses $C_1, \ldots, C_i$ can potentially be eliminated as well.

Implementation Our first optimisation is to transform the DL program into an equivalent filtered form. To obtain a filtered program we use an iterative process. Here we start from the initial DL program and we eliminate as many clauses as we can. However, if we successfully eliminated the last remaining clause of at least one predicate, then we start the whole process again. We do as many iterations as needed to reach a fixpoint, i.e. to have a set of clauses from which we cannot eliminate any more clauses.

Example As an example for filtering, let us consider the DL program of the Happy problem presented in Figure 2.8. In the first iteration, we can eliminate clauses in lines 4–5 and 7–8 as they invoke the orphan goal $not_{\text{Happy}}(B)$, and there is no way to reach these clauses from predicate $\text{Happy}/1$.

As these were the last clauses of their corresponding predicates, $not_{\text{Clever}/1}$ and $not_{\text{Pretty}/1}$ have actually become orphans. Therefore, we apply one more iteration. Now we cannot eliminate anything else: we reached a fixpoint, having the only clause $\text{Happy}/1$ in lines 1–2 (and the ABox facts in line 10).

2.4.3 Classification

Within the filtered DL program we distinguish between different groups of predicates based on their properties. This classification turns out to be useful when generating the Prolog programs as it provides guidelines for what to generate and it also serves as a basis for further optimisations.

Predicate classes We classify a predicate into one of the following four main groups.

1. A predicate $P$ is atomic if all of its clauses have empty bodies. Examples for atomic predicates are $\text{Clever}/1, \text{Pretty}/1$ and $\text{hasChild}/2$ in the Happy and Iocaste DL programs.

2. $P$ is a query predicate if it is not atomic and it satisfies the following three conditions:

   (i) $P$ is not recursive
   (ii) $P$ is not reachable from $not_P$ (i.e. $P$ is not DNR)
   (iii) every predicate that is referred to in any of the clauses of $P$ is either an atomic or a query predicate

   An example of a query predicate is $\text{Happy}/1$ in Figure 2.8.
3. The notion of orphan predicate has already been introduced in Section 2.3.5. Examples include predicates `not_Ans/1` and `not_Happy/1` in Figures 2.7 and 2.8.

4. Finally, a predicate $P$ is a general predicate if it is neither atomic, nor query, nor orphan. A general predicate $P$ can be further classified into subgroups based on whether $P$ is recursive, is of type ANR or DNR. The general predicates in the Iocaste knowledge base are the following: `Ans/1` (not recursive, not DNR, ANR), `Patricide/1` (recursive, not DNR, not ANR) and `not_Patricide/1` (recursive, not DNR, not ANR).

**Implementation** A predicate $P$ is classified as atomic or orphan simply by checking whether the specified condition holds for $P$.

However, to determine the set of query predicates, we use an iterative process similar to the one used in filtering (cf. Section 2.4.2). The idea is that we iterate as long as we find at least one new query predicate. We note that we actually use one single iterative process which encapsulates filtering as well as query predicate classification.

After this, all the remaining predicates are considered to be general.

**Use of classification information** Having classified the predicates of a DL program we can apply specific compilation schemes for certain classes. In the following we examine each of the predicate classes.

- **atomic predicates**: Atomic predicates directly correspond to tables in a database and thus their translation does not require an extra argument for the ancestor list.

- **query predicates**: The conditions in the definition guarantee that in the case of a query predicate $P$, we (i) do not need to check for loops, (ii) do not need to apply ancestor resolution, and (iii) do not need to pass the ancestor list to any of the goals in the body of $P$.

A further consequence of these facts is that in the code generated for query predicates we have only one argument, similarly to atomic concept predicates.

Now we can also explain the naming: it comes from the fact that query predicates can directly be transformed into database queries. This is because the execution of query predicates do not need any further processing (e.g. ancestor resolution) on the Prolog side.

- **orphan predicates**: In most of the cases we do not need to pass the new ancestor list to orphan calls, but the old one. Namely, as an orphan goal $G$ can only succeed if the ancestor list contains $G$ negated, we only have to pass the new ancestor list if the goal from which we call $G$ is the negation of $G$. This is not the case, for example, in lines 10 and 16 in Figure 2.11, so we could optimise the code in Figure 2.11 by passing the old ancestor list (stored in variable $B$) to the orphan calls `not_Ans`.

- **general predicates**: We need to generate loop tests only for recursive, and ancestor tests only for general DNR predicates. Updating the ancestor list is only required for ANR predicates.

**Examples** We now show some examples of how the Prolog code can be simplified due to classification. Since `Happy/1` in Figure 2.8 is classified as a query predicate, after the filtering step, we can further simplify the Happy program resulting in the code shown in Figure 2.14. Note that we have actually obtained one of the simplest translations for the Happy problem.

```
Happy(A) :- hasChild(A, B), hasChild(B, C), hasChild(B, D),
           Clever(C), Pretty(D).
Clever(lisa). Pretty(lisa). hasChild(kate, bob). hasChild(bob, lisa).
```

Figure 2.14: The Happy program after filtering and classification.
We also note that in the case of the Iocaste program in Figure 2.11, classification directly results in omitting lines 1, 2, 7 and 13. Lines 1 and 2 can be removed because predicate Ans/1 is classified as a non-recursive and non-DNR general predicate. Ancestor tests in lines 7 and 13 can be omitted as Patricide/1 and not_Patricide/1 are non-DNR predicates.

2.4.4 Body ordering

An important optimisation is to order the goals in the body of the generated clauses so as to minimise the execution time. This is a generic idea used in some form or other by many systems. For example, in the case of databases, query optimisation is an essential task, as without it we would not be able to answer complex queries [39]. Query optimisation is similarly important when querying non-relational information sources, such as XML [35].

Query optimisation often relies on statistical information, such as the size of database tables or the number of distinct values in a given column. In the present work we do not take into account such information and so we restrict our attention to optimisations which consider only the TBox part of the DL programs.

Prolog systems also use body reordering. For example, the Mercury compiler reorders the conjunctions in clauses for more efficient execution [112]. Body reordering, instantiation analysis and related techniques are used by many parallel systems as well. For example, in the Andorra [24] system the deterministic goals in a clause are moved to the front.

In our case we have special clauses to work with, as described in Section 2.3.2. Recall, for example, that binary predicates contain all the variables of a clause. We also have special (e.g. orphan) predicates to deal with. Altogether, these peculiarities allows us to use a simple, specialised ordering technique.

Below we first propose a possible ranking between the different kinds of goals in a body. This ranking uses heuristics applicable for DL programs. After this we introduce the simple algorithm we use for body ordering. Note that this algorithm is actually only the first step, as it forms the basis of a more complex body restructuring technique described in Section 2.4.7.

Ranking of goals

Let us start with considering some principles for ranking. Atomic and query predicates can be answered by using ABox facts only, i.e. they correspond to (maybe complex) database queries. On the other hand, general predicates, such as Patricide/2 for example, may require complex, possibly recursive, execution on the Prolog side. According to this, our first heuristic is to invoke atomic and query predicates before general predicates.

Naturally, the instantiation of the predicates also plays an important role. It is worth invoking a fully ground role at a given point rather than a role with an uninstantiated variable. The former simply checks whether a relation holds between two individuals. The latter enumerates all possible pairs of individuals for which the given relation hold, leaving a possibly huge choice point behind.

We have also found that given two roles with uninstantiated variables it is usually better to invoke first the one with a head variable, i.e. the variable in the head of a clause. The justification behind this step is that once we have selected a role that instantiates the head variable we have actually reduced the problem of instance retrieval to instance checking, for the given concept.

A further issue to discuss is the place of the orphan goals within a body. Recall that orphan goals can only succeed by ancestor resolution. For example, the orphan goal not_Happy(D, B) in line 9 in Figure 2.12 can only succeed if the ancestor list B contains a Happy/1 ancestor. However, in the case of a successful execution, we cannot be sure that variable D will be instantiated. This is because an orphan call can succeed by unifying its head variable with a variable in the ancestor list.

This property of the orphan goals suggests to put them in the first available place where they are ground. However, it also seems to be a good idea to move an orphan goal to the very front in a body. This is because orphan goals tend to fail very often: if they are placed at the front, in the case of failure, we do not need to execute the rest of the clause.

Note, however, that placing orphan goals to the front invalidates the proof of Proposition 11 on page 23 as now the case (c2) can also happen. Fortunately, Proposition 15 ensures that ancestor resolution remains deterministic for DL programs, even if the invocations of the orphan goals are moved to the front of a body.
Based on the above discussion, a possible ranking order, called *base ranking*, is summarised in Figure 2.15. Here we define 9 categories of goals and give orphan goals the maximum priority. If there are more goals within the same category, the selection between them is unspecified, i.e. any of them can be chosen. For example, if we have two non-ground atomic concepts, either of these can come first. Note that the base ranking ensures the binary first rule introduced in Definition 5, except for orphan goals.

```
<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>orphan goal</td>
</tr>
<tr>
<td>ground role</td>
</tr>
<tr>
<td>ground atomic or query concept</td>
</tr>
<tr>
<td>role with 1 unbound variable</td>
</tr>
<tr>
<td>role with 2 unbound variables, but at least one of them is a head variable</td>
</tr>
<tr>
<td>role with 2 unbound variables</td>
</tr>
<tr>
<td>non-ground atomic or query concept</td>
</tr>
<tr>
<td>ground general concept</td>
</tr>
<tr>
<td>non-ground general concept</td>
</tr>
</tbody>
</table>
```

Figure 2.15: Base ranking: a possible ranking of the different types of goals within a body.

**The ordering algorithm**

Here we specify a simple algorithm that orders the body of a clause within a DL program as shown in Figure 2.16. This algorithm has three inputs: the body to be ordered (*B*), a pre-defined ranking of the different kinds of goals (*R*) and an initial variable list (*V*) containing those variables that are known to be instantiated at the beginning.

```
1. input parameters: *B*, *R*, *V*, *i* := 1
2. if *B* = ∅ exit with *G*₁, . . . , *G*ᵢ₋₁
3. *G*ᵢ := best goal in *B* according to ranking *R* wrt. variables *V*
4. *B* := *B* \ {*G*ᵢ}
5. if *G*ᵢ is non-orphan *V* := *V* \ variables of goal *G*ᵢ
6. *i* := *i* + 1
7. goto step 2
```

Figure 2.16: The ordering algorithm used to optimise the execution of a body.

The idea is to always select the best goals one by one, to be placed in the final body (step 3). To be able to assess the groundness of arguments we keep track of the set *V* of variables instantiated so far. *V* is initialised as a parameter (step 1) and is updated to include the variables of the best goal selected (step 5). Having selected a goal, we continue by iteratively ordering the rest of the body (step 7).

As an example, reordering the body of the main clause of *Patricide/2* yields the clause presented in Figure 2.17 (cf. line 9 in Figure 2.11). Here the orphan call *not_Ans/2* is moved to the front. The second goal is a role containing a head variable. The third one is also a role with at least one variable instantiated: the instantiation of *D* is unknown at compile-time. Finally, the last goal is a ground general concept call.

```
Patricide(A, B) :- C = [Patricide(A)|B], not_Ans(D, B), hasChild(E, A),
                    hasChild(D, E), Patricide(E, C).
```

Figure 2.17: Reordered version of the main clause of *Patricide/2*. 

34
For another example let us consider clause Happy/1 in Figure 2.14. Using body reordering on this clause we get the clause presented in Figure 2.18. Note that goal Clever(C) is now placed where it first becomes ground.

```
Happy(A) :- hasChild(A, B),
           hasChild(B, C), Clever(C),
           hasChild(B, D), Pretty(D).
```

Figure 2.18: Reordered version of clause Happy/1.

### 2.4.5 Multiple argument indexing

We can notice that goal has_child(D, E) in line 2 in Figure 2.17 is always called with the second argument instantiated. If we use a database system to store the content of the ABox this call can be efficiently executed. This is because databases can do indexing on every column of a table.

In most Prolog systems, however, indexing is done only on the first argument of the head. This may raise performance issues if we use Prolog for storing large amounts of ABox facts.

To achieve multiple argument indexing in the generated programs we do the following. For each role \( P \) we generate a new role \( \text{idx}_P \). This new set of Prolog facts (called index predicate) captures the inverse relation between the arguments of \( P \), i.e. \( \text{idx}_P(X, Y) \) holds if and only if \( P(Y, X) \) holds. In the case of the Iocaste problem this effectively means that we add the following index predicate to the generated program:

```
1. idx_hasChild(o, i).
2. idx_hasChild(p, Y) :- idx_hasChild_p(Y).
3. idx_hasChild(t, p).
4. idx_hasChild_p(i).
5. idx_hasChild_p(o).
```

Now, we change every invocation of role \( P \) to \( \text{idx}_P \) if the second argument of \( P \) is instantiated, but the first is (possibly) not. Thus, the ordered clause for Patricide/2 in Figure 2.17 takes the following form:

```
Patricide(A, B) :- C = [Patricide(A)|B], not_Ans(D, B), hasChild(E, A),
                  idx_hasChild(E, D), Patricide(E, C).
```

Note that we do not actually have to generate index predicates for every role in the ABox, because using compile time analysis we can identify those roles \( P_1, \ldots, P_i \) that need indexing at all (i.e. they are called at least once in such a way that their second argument is ground, but the first is possibly not).

Also note that, most Prolog implementations create a choice point when both arguments of a role \( P \) are instantiated, although such invocations can only succeed once (because an ABox cannot contain a given \( P(i,j) \) axiom twice). For example, consider the goals hasChild\( i, o \) or idx_hasChild\( p, i \).

As a further optimisation, to avoid these choice points, we apply the commonly known technique of using auxiliary predicates. Namely, given a role predicate \( P \) (including the above introduced index predicates) with facts \( F \) we do the following. For every maximal set \( D \subseteq F \) of facts, which share their first argument we introduce a single grouping clause \( N(A, Y) : \neg T(Y) \). Here, \( N \) is the name of predicate \( P \), \( Y \) is a variable and \( A \) is the constant shared by all of the facts in the first argument position in \( D \). \( T \) is the name of a newly introduced predicate containing facts \( T(Z_1), \ldots, T(Z_k) \) that correspond to the constants in the second arguments of the facts in \( D \), i.e. \( \forall i. D_i(A, B_i) \in D \rightarrow Z_i = B_i \).

As an example, consider the optimised version of the four clauses of the predicate idx_hasChild/2 introduced above.

```
idx_hasChild(o, i).
idx_hasChild(p, Y) :- idx_hasChild_p(Y).
idx_hasChild(t, p).
idx_hasChild_p(i).
idx_hasChild_p(o).
```
Here, in line 3, we use the grouping clause \texttt{idx\_hasChild(p, Y)} that enables Prolog not to create any choice points when invoking the goal \texttt{idx\_hasChild(p, i)} or \texttt{idx\_hasChild(p, o)}.

### 2.4.6 Ground goal optimisation

An important optimisation step is to make sure that the truth value of ground goals, i.e. goals with only instantiated arguments, is calculated only once. Note that by default this behaviour is not provided by the Prolog execution, but is supported, for example, by Mercury [112].

To achieve this, we duplicate a general or query predicate \( P \), i.e. we create two versions of \( P \) depending on whether we assume that the head variable is instantiated or not. These variants of \( P \) are called non-deterministic (\texttt{nondet}) and deterministic (\texttt{det}) variants, respectively.

We also create a \texttt{choice} predicate for the general case that checks if the head variable is ground at runtime. This predicate then calls either the \texttt{nondet} or the \texttt{det} variant of predicate \( P \).

The differences between the variants of a predicate \( P \) are the following:

1. We place a Prolog cut (denoted by \(!\)) at the end of each clause in the \texttt{det} variant. This results in pruning the rest of the search space after a successful execution of the \texttt{det} variant.

2. We order the body of the clauses in the \texttt{det} variant based on the assumption that the head variable \( H \) is instantiated (i.e. the ordering algorithm in Figure 2.15 is executed with the initial variable list \( V = \{H\} \)).

Finally, we transform every goal in the program that calls a general or query predicate \( P \) into another goal that calls \texttt{choice}\_\( P \) instead. This technique is illustrated in Figure 2.19.

```prolog
choice\_Patricide(A, B) :-
  ( nonvar(A) -> det\_Patricide(A, B)
  ;   nondet\_Patricide(A, B) )
.

nondet\_Patricide(A, B) :- member(C, B), C == Patricide(A), !, fail.
nondet\_Patricide(A, B) :- memberchk(not\_Patricide(A), B).
nondet\_Patricide(A, _) :- Patricide(A).
nondet\_Patricide(A, B) :- C=[Patricide(A)|B], not\_Ans(D, B), hasChild(E,A),
                          idx\_hasChild(E, D), det\_Patricide(E, C).

det\_Patricide(A, B) :- member(C, B), C == Patricide(A), !, fail.
det\_Patricide(A, B) :- memberchk(not\_Patricide(A), B), !.
det\_Patricide(A, _) :- Patricide(A), !.
det\_Patricide(A, B) :- C=[Patricide(A)|B],not\_Ans(D,B), idx\_hasChild(E,A),
                          idx\_hasChild(E, D), det\_Patricide(E, C), !.
```

Figure 2.19: Two variants of predicate \texttt{Patricide/2}

Note that in lines 10 and 16, instead of \texttt{choice}\_\texttt{Patricide/2}, we directly invoke predicate \texttt{det}\_\texttt{Patricide/2}. This is a further optimisation step. Namely, we can directly call the \texttt{det} variant of a predicate \( P \) if we know already at compile-time that the first argument of the specific invocation of \( P \) is ground. In our case we can be sure that variable \( E \) is instantiated at the time of calling \texttt{det}\_\texttt{Patricide/2}, because the predicate call \texttt{idx\_hasChild(E, D)} instantiates it.

Also note the difference between the body goals in the two variants in Figure 2.19 (lines 9–10 and 15–16). In the \texttt{det} variant we assume that variable \( A \) is instantiated at call time, therefore we use the \texttt{idx} variant of the goal \texttt{hasChild(E, A)}.  

36
2.4.7 Decomposition

The goal of decomposition is to split a body into independent components. This is achieved by uncovering the dependencies between the goals of the body. This process, on one hand, introduces a higher level body ordering, where the independent goal groups are ordered first, and then the individual groups are split and ordered recursively. More importantly, the discovery of independent components makes it possible to use a generalisation of the ground goal optimisation, by applying this technique to a whole independent goal group. For DL programs generated from a $SHIQ$ KB, this practically means recovering certain useful structural properties of the initial TBox axioms. Before we go into details we show an example to demonstrate a problem which can be solved using decomposition.

Introductory example

Let us recall clause $Happy/1$ shown in Figure 2.18 where we consider someone happy if she has a child having both a clever and a pretty child. Although the body of this clause is ordered according to our base ranking, in certain cases the execution of it is far from optimal. For example, consider the ABox specified below.

```
hasChild(kate, bob).
hasChild(bob, lisa_1).
hasChild(bob, lisa_2).
...
hasChild(bob, lisa_n).
Clever(lisa_1).
Clever(lisa_2).
...
Clever(lisa_n).
```

Here we know that $bob$ is the child of $kate$ and he has $n$ clever children (denoted by $lisa_1, ..., lisa_n$). Note, however, that nobody is known to be pretty. Thus, with respect to this ABox, we cannot conclude that $kate$ is happy, i.e. the invocation...

```
?- 'Happy'(kate).
```

...fails. The problem is that, before this happens, we examine lots of useless branches of the search space. Namely, we enumerate all children of $bob$ and check whether they are clever. We do this in spite of the fact that $bob$ has no pretty children at all, even though, having a pretty grandchild is a necessary condition for $kate$ being happy. What happens is that we explore the choice point created in line 2 in Figure 2.18, although goals in line 3 are bound to fail.

Note that this behaviour would not change if we applied ground goal optimisation here, i.e. if we used the $det$ variant of the clause (cf. Section 2.4.6) in Figure 2.18. The order of the goals in the body would be the same. The cut at the end of the clause would not matter either, as the goal $Happy(kate)$ fails.

What we need here is the realisation that the $hasChild(B, C)$, $Clever(C)$ group of subgoals, used for checking that $bob$ has a clever child, is independent from the remaining subgoals of the body. Thus, once we have proved that $bob$ has a clever child, there is no point in proving this property in other ways.

The solution

In this specific example, the real reason behind the inefficient execution is that during the Prolog translation we do not utilise the structural properties of the TBox axiom in Figure 2.5. This axiom actually describes that somebody is happy if she has a child satisfying a certain condition, let us call it “special”. Here somebody is “special” if she has a clever child as well as a pretty child. This condition can be split into two independent parts $hasChild(B, C)$, $Clever(C)$ and $hasChild(B, D)$, $Pretty(D)$ sharing only a single variable $B$. Assuming that $B$ is ground, we can stop enumerating her children once a clever one is found, as further choices of variable $C$ do not affect the remaining goals.
The solution is to incorporate this knowledge by decomposing the body of clause Happy/1 in a certain way, as shown in Figure 2.20.

```prolog
Happy(A) :-
    hasChild(A, B),
    ( hasChild(B, C),
        Clever(C) -> true
    ),
    ( hasChild(B, D),
        Pretty(D) -> true
    ).
```

Figure 2.20: Decomposed version of the Happy/1 clause.

Here the clause starts with a single goal that represents the knowledge that somebody should have at least one child in order to be happy (line 2). This child should be “special”, which condition is captured by the two consecutive components in the clause (lines 3–5 and 6–8). The idea here is that we only look for the first solution of these components, i.e. we place an implicit Prolog cut (by using the conditional expression operator ->) into these. This ensures that once a component succeeds it prunes the rest of its search space. This is, in fact, the ground goal optimisation, applied to a whole component, rather than to a single goal.

Note that the goal hasChild(A, B) in line 2 generates a choice point, which we cannot eliminate here as we cannot be sure that B is “special”. On the other hand, if we used ground goal optimisation (cf. Section 2.4.6) then the cut (!) at the end of the det variant clause would prune this choice point.

The process of decomposition

The decomposition process relies on identifying independent components in clause bodies, i.e. subgoal sequences that do not share uninstantiated variables. Such techniques have been extensively studied, mostly in the context of parallel execution of logic programs, for example in [101].

Because of the properties of DL predicates here we can apply a very simple special algorithm. The decomposition process is actually a modification of the ordering algorithm introduced in Figure 2.16. Namely, we provide three more steps (2a) – (2c) as shown in Figure 2.21.

2a. split B into partition \{B_1, \ldots, B_n\} wrt. V
2b. if \(n = 1\) continue at 3
2c. recursively apply this algorithm (from step 1) for (\(B_1, R, V\)) resulting in \(C_1, \ldots, (B_n, R, V)\) resulting in \(C_n\) and return \(G_1, G_{i-1}, c(C_1), \ldots, c(C_n)\)

Figure 2.21: The decomposition process (extension of the algorithm in Figure 2.16)

Decomposition starts with step 2a which partitions the set of body goals into one or more subsets in such a way that goals in different partitions share only variables in \(V\) (the set of variables considered to be instantiated) and the maximal number of partitions is obtained. If the decomposition has been unsuccessful, i.e. a single partition is returned, then we continue with the normal goal ordering algorithm (step 2b).

If multiple partitions have been obtained then each of these is ordered and recursively decomposed (step 2c). In this case the output of the modified ordering algorithm contains the goals collected so far, followed by the components. The latter are distinguished from ordinary goals by being encapsulated in a \(c(\ldots)\) structure. This marks the independent units where pruning can be potentially applied.

Note that the components themselves may also undergo an ordering phase, but this is not detailed here.
We illustrate the idea of recursive decomposition on the nondet variant of clause Ans/1 from the Iocaste problem. This is shown in Figure 2.22. Note that the body of this clause forms a single component, which can only be decomposed further by selecting a primary goal, i.e. the goal in line 3.

Also note that variables used for ancestor resolution in the generated program are not considered during the decomposition process as it is performed on the DL program directly. This is the reason why goals in line 5 and line 8 can be placed into separate components, although both of them contain variable C.

```
nondet_Ans(A, B) :-
    C=[Ans(A) | B],
    hasChild(A, D),
    { hasChild(D, E),
      det_not_Patricide(E, C) ->
        true
    },
    det_Patricide(D, C).
```

Figure 2.22: Transformation of clause Ans/1 using decomposition.

### 2.4.8 Projection and supersets

As we have seen in the previous section, decomposition helps to reduce the number of useless choice points in a body by placing conditional structures at the appropriate places. However, the fact that we cannot directly eliminate the choice point created by the first component in the nondet variant of a clause can cause serious performance problems. Below we first demonstrate this problem through an example, then we introduce the notion of projection and superset used in our solution.

**Example**

Let us consider the decomposed version of clause Ans/2 shown in Figure 2.22 and examine how this clause is executed if we have a fairly large Iocaste pattern in our ABox such as the one shown in Figure 2.23.

![Iocaste Pattern](image)

Let us assume that goals in lines 3 and 4 in Figure 2.22 succeed with variable substitutions \( \lambda=i, D=\sigma, E=p_1 \). The next goal checks whether not_Patricide(p_1) holds wrt. the given set of ancestors (line 5). This goal succeeds using the clause in lines 15–16 in Figure 2.11. Note however, that for this we actually have to prove that not_Patricide(p_2) holds. This, in turn, is reduced to the question whether not_Patricide(p_3) holds, etc. Finally we reach the last individual in the chain named t which is directly known to be not patricide. After this, the last goal in Figure 2.22 is checking if \( \sigma \) is a patricide. This can be directly seen from the ABox itself (line 8), so we have just successfully proved that Ans(X) holds in case of X=i.
The problem comes after we ask for more solutions. Note that thanks to decomposition and ground goal optimisation, the only choice point at the moment is due to the goal hasChild(A, D) in line 3. This goal will sooner or later enumerate in D all the remaining children of i: p₁, ..., pₙ. Now, if D=p₁, for example, then we have to prove again that hasChild(p₁, p₂) and not_Patricide(p₂) hold. We can easily see that this way we actually traverse the chain n times making altogether O(n²) steps to prove that i is a solution. As a matter of fact, we will actually obtain i as a solution n times, something which should be avoided.

**Working with supersets**

We can eliminate the use of the nondet variants of the predicates by calculating a so called superset of individuals. Once we have only deterministic predicates, the above mentioned problems are automatically solved.

In the following first we define the notion of superset, then we show how it can be used to eliminate the non-deterministic predicates from the generated programs.

**Definition 23.** Let $I(P)$ denote the set of solutions of a unary predicate (clause) $P$ wrt. a Prolog program. By a solution of a clause $C$ we mean a solution of the predicate that $C$ belongs to, such that in the corresponding proof tree the root and its children correspond to clause $C$.

A set of instances $S$ for which $I(P) \subseteq S$ holds is called a superset of predicate (clause) $P$.

According to the definition, the superset of a predicate is a set of instances that contains all the solutions of the predicate (and possibly some other individuals as well). For example, the set of individuals \{i, o, t\} forms a superset of predicate Ans/1, as it contains individual i.

Now, given a predicate $P$ and one of its supersets $S$ we can eliminate the nondet variant of $P$ as follows: we create a new predicate that invokes the det variant of $P$ for each individual $i \in S$. Technically, this logic can be built in the choice predicate, as exemplified below:

```prolog
choice_Ans(A, B) :-
    ( nonvar(A) -> det_Ans(A, B)
    ; member_of_superset_Ans(A), det_Ans(A, B)
    ).
...```

Here we call the det variant directly if $A$ is instantiated (line 2). However, we also call the det variant if $A$ is uninstantiated, after enumerating in $A$ the elements of the superset (line 3).

Note that this technique has an important property: it ensures that every solution is returned only once, e.g. invoking choice_Ans(A, []) enumerates instance i only once wrt. the usual Iocaste ABox. Also note that the above schema can be used for supersets that do not fit into memory: member_of_superset_Ans(A) can be a database invocation that enumerates the individuals in the superset.

In the following we describe an algorithm that specifies a set of instances for a predicate $P$, then we show that this set is actually a superset of $P$.

**Calculating supersets**

We focus our attention on how to create a “good” superset for a given unary predicate. A superset is “good” if (1) it contains not many false individuals and if (2) it is easy to calculate. A superset containing every individual of the given ABox is practically useless as this way we get back to the traditional ABox reasoning approach where we do an instance check for each possible individual (cf. Section 2.2). It is also true that we do not want to spend much time building the superset: the whole idea of using a superset is to have more efficient execution than without it.

**Definition 24.** The projection of a role predicate $P$ wrt. its $n$th ($n = 1, 2$) argument is $Pr_n(P) = \{ v_n | (v_1, v_2) \in I(P) \}$. The projection of a concept predicate $C$ wrt. its only argument is $Pr_1(C) = I(C)$. 

40
This definition says that the projection of a predicate is a set of instances. In the case of a concept predicate this set is equal to the set of the solutions of the predicate. In the case of a role, projection returns the set of instances that can appear in the specified argument position.

For example, \( Pr_{1}(\text{hasChild}(A, B)) \) wrt. the usual Iocaste knowledge base is the set \{i, o, p\} excluding \( t \); \( t \) does not have any children. Note that this projection can be calculated by the \( \text{setof}(\text{A, B}^\text{\text{hasChild}}(A, B), \, R) \) Prolog call. Now we present the definition of the projected label of a clause that is either a set of instances or a functor.

**Definition 25.** Let \( C \) be a unary clause and let \( W \) be the set of all atomic and query goals in \( C \) which contain the head variable. We define the projected label of \( C \), denoted by \( Pl(C) \), as follows.

If \( C \) is a fact of form \( C(a) \) then \( Pl(C) \) is the set \{\( a \)\}. Otherwise, if \( W \neq \emptyset \), then \( Pl(C) \) is calculated as the intersection of the projections of the goals in \( W \) wrt. the head variable, i.e. \( Pl(C) = \cap_{G \in W} Pr_{p_{i}}(G_{i}) \), where \( p_{i} \) is the position of the head variable in goal \( G_{i} \).

Finally, if \( W = \emptyset \), then \( Pl(C) \) is the functor of the general predicate in \( C \) which comes first wrt. the standard Prolog term ordering.

For example, let us consider the Iocaste program presented in Figure 2.7. The projected label of the clause in lines 1–2 is the set \( Pr_{1}(\text{hasChild}(A, B)) \), while the projected label of the clause \( \text{not_Patricide}/1 \) in lines 7–8 is \( Pr_{1}(\text{hasChild}(A, B)) \cap Pr_{2}(\text{hasChild}(E, A)) \). The projected label of a clause not containing any atomic or query predicates is a functor. For example, if \( C \) is \( p(X) \leftarrow q(X), \, r(X) \), then \( Pl(C) = q/1 \). Here we assumed that \( q/1 \) and \( r/1 \) are general predicates.

Using the definition of the projected label, we introduce miniset graphs which we will use to define the notion of the miniset of a predicate.

**Definition 26.** Let \( S \) be a DL program. Let \( P \) be a predicate in \( S \) with clauses \( C_{1}, \ldots, C_{k} \). The **miniset tree** of \( P \) is a labelled tree \( T(P) = (V, E, L) \) as defined in the following. The nodes of \( T(P) \) correspond to predicate \( P \) and its clauses, i.e. \( V = \{P, C_{1}, \ldots, C_{k}\} \). There are edges between \( P \) and its clauses, i.e. \( (P, C_{i}) \in E, \, 1 \leq i \leq k \).

Node \( P \) is labelled with the functor of \( P \). A node corresponding to a clause \( C \) is labelled with its projected label, i.e. \( (C, Pl(C)) \in L \).

Now, let us consider all the miniset trees \( T(P) \), where \( P \) is a predicate of \( S \). We call the union of these trees the **miniset graph** of DL program \( S \), denoted by \( M(S) \). Here by union we mean the usual graph union extended with the idea that nodes with the same label are merged into a single node.

According to the definition, a miniset graph contains labels of two types: containing a predicate functor or a set of individuals. As an example, let us consider the miniset graph of the Iocaste program in Figure 2.24.

![Figure 2.24: The miniset graph of the Iocaste DL program in Figure 2.7.](image)

Now we are ready to precisely formulate the definition of the miniset of a predicate.

**Definition 27.** Let \( G \) be a miniset graph. The **miniset** \( M(P) \) of a predicate \( P \) is calculated as the union of the sets of individuals that can be reached from node \( P \) in graph \( G \).

For example, the miniset of predicate \( \text{Patricide}/1 \) in the Iocaste knowledge base is \( \{o\} \cup \{o, \, p, \, t\} = \{o, \, p, \, t\} \). Note that such minisets can be efficiently calculated at the database level as the projected labels in the miniset graph are composed only of atomic and query predicates.
Proposition 16. For an arbitrary predicate \( P \), \( M(P) \) is a superset of \( P \).

Proof. Let predicate \( P \) of a DL program \( S \) have clauses \( C_1, \ldots, C_n \). By definition we now that \( I(P) = \bigcup_{C \in P} I(C) \), i.e. the solutions of a predicate equals to the union of the solutions of its clauses. We also know that for a clause \( C \) where \( Pl(C) \) is a set of individuals \( I(C) \subseteq Pl(C) \) holds. This is an obvious conclusion of Definition 25. Now we only need to consider the clauses of \( P \) which do not fall into the above category. Let \( D \) be such a clause of \( P \), i.e. \( Pl(D) \) is a functor \( F \). This functor corresponds to a node in \( M(S) \). By simple induction it follows that \( I(D) \subseteq M(F) \). 

We just proved that the miniset of a predicate \( P \) calculated according to Definition 27, is a superset of \( P \).

Implementation

In our implementation we calculate the miniset of a predicate \( P \) in the following way. First we collect the conjunction of goals from the clauses of \( P \) corresponding to the projected labels reachable in the miniset graph from \( P \). For example, we determine the following conjunctions from a fictitious predicate having three clauses (the head variable is \( X \)):

\[
\begin{align*}
1 & \text{hasChild}(Y, X), \text{hasChild}(X, Z), \text{hasFriend}(X, W) \\
2 & \text{hasChild}(X, Y) \\
3 & \text{Rich}(X)
\end{align*}
\]

From the union of lines 1–3 we can calculate the members of the superset by an appropriate Prolog `setof` and `member` goals:

\[
\begin{align*}
\text{member_of_superset_goal}(A) & :- \text{setof}(X, \text{goal}(X), S), \text{member}(A, S). \\
\text{goal}(X) & :- \\
& \text{hasChild}(Y, X), \text{hasChild}(X, Z), \text{hasFriend}(X, W); \\
& \text{hasChild}(X, Y); \\
& \text{Rich}(X).
\end{align*}
\]

Note that we can simplify `goal/1` above: we can omit line 4 as it is overridden by the more general goal in line 5. Namely, \( Pr_1(\text{temp}(X)) \subseteq Pr_1(\text{hasChild}(X, Y)) \) holds, where \( \text{temp}(X) \) is a goal invoking the temporary clause shown below:

\[
\begin{align*}
\text{temp}(X) & :- \text{hasChild}(Y, X), \text{hasChild}(X, Z), \text{hasFriend}(X, W).
\end{align*}
\]

Evaluating the appropriate `setof` goal can actually be done at compile time as well as runtime. The former case corresponds to some kind of ABox preprocessing which can be useful for performance reasons in certain applications. The latter case means the minisets are calculated by the generated program on-the-fly. In the DLog system we apply the latter approach. If the ABox is stored in a database then we directly use it to enumerate the instances of the superset.

The way how we calculate the superset of a predicate \( P \) above is sound, although it can be further optimised. Note that when determining the superset of \( P \) we can utilise the fact that \( \text{choice}_P \) is called only once during the whole execution of a query: we can consider \( \text{choice}_P \) as an entry predicate, where the ancestor list is known to be empty.

We use this fact in our implementation to provide a small, but useful optimisation: we omit those clauses of \( P \) from the calculation of the superset of \( P \) which contain orphan goals that always fail, because of the initially empty ancestor list.

Namely, we use a list to store the orphan goals that can possibly succeed. In the beginning, this list contains only a single element: the negation of \( P \). Whenever we reduce calculating the miniset of a clause \( C \) to recursively calculating the miniset of predicate \( T \) (i.e. when we go through a predicate in the miniset
graph), we extend our list with the negation of \( T \). If any time we process a clause \( C \) that contains an orphan goal not present in the list, we simply skip \( C \).

To conclude the subsection and to demonstrate this optimisation step, in Figure 2.25 we show the translation of the Iocaste problem containing all the optimisation we have discussed so far. Notice how simple is the entry predicate for \( \text{Patricide} \) (lines 15–16): it only invokes the atomic ABox predicate. This is because the superset of \( \text{Patricide/2} \) contains only the single atomic call \( \text{Patricide}(A) \): we have skipped the other clause of \( \text{Patricide/2} \) as it contains an orphan goal that cannot succeed when \( \text{Patricide} \) is the entry predicate. Because of this, we do not even generate the conditional structure usually present in choice_ predicates.

```
choice_Ans(A, B) :-
    { nonvar(A) -> det_Ans(A, B)
    ; setof(A, C^hasChild(A,C), D), member(A, D), det_Ans(A, B)
    }.

det_Ans(A, B) :- member(C, B), C==Ans(A), !, fail.
det_Ans(A, B) :- C = [Ans(A)|B],
    hasChild(A, D),
    { hasChild(D, E),
        det_not_Patricide(E, C) ->
        true
    },
    det_Patricide(D, C), !.

choice_Patricide(A, _) :-
    Patricide(A).
```

Figure 2.25: The final Prolog translation of the Iocaste problem.

### 2.4.9 Transforming role axioms

Our last optimisation involves role axioms. We present here a compilation scheme for \( \mathcal{SHIQ} \) roles which is more efficient than the one introduced in Section 2.3.5. We note that the indexing technique introduced in Section 2.4.5 slightly modifies the scheme described in this section. These modifications are mainly technical and not covered here.

The idea is that we would like to avoid the usage of loop elimination in the case of clauses with binary literals in the head as loop checking is an expensive operation. However, including role axioms without any modifications into the resulting Prolog code may easily lead to infinite loops. For example, the naive transformation of the knowledge that \( R \) and \( S \) are equivalent roles yields the following piece of program:

```
R(A, B) :- S(A, B).
S(A, B) :- R(A, B).
```

To define a safe and efficient translation for role axioms, we first start with some auxiliary definitions.

**Definition 28.** Let \( G \) be an arbitrary directed graph. The reduced graph of \( G \), denoted by \( G_r \), is defined as follows. The vertices of \( G_r \) are the strongly connected components (SCC) of \( G \). There is an edge in \( G_r \) from \( A \) to \( B \) if and only if there is an edge in \( G \) from one of the vertices in the SCC corresponding to \( A \) to one of the vertices in the SCC corresponding to \( B \).
Definition 29. Let $R$ be a role name. The inverse role name of $R$, denoted by $\text{Inv}(R)$, is defined as follows.

$$\text{Inv}(R) = \begin{cases} \text{S} & \text{if } R = \text{inv}_S \\ \text{inv}_R & \text{otherwise} \end{cases}$$

For example, the inverse of role hasChild is $\text{inv\_hasChild}$, while the inverse of role $\text{inv\_hasChild}$ is hasChild.

Definition 30. Let $S$ be a set of role names. $C$ is called the inverse closure of $S$ if $C$ is the smallest set of roles such that $R \in S \rightarrow R \in C$ and $R \in C \rightarrow \text{Inv}(R) \in C$.

Accordingly, the set $\{\text{hasChild}, \text{inv\_hasChild}\}$ is the inverse closure of the set $\{\text{hasChild}\}$. We now define the concept of role dependency graph which will form the basis of the efficient transformation of roles.

Definition 31. For a given ABox $\mathcal{A}$ and RBox $\mathcal{R}$ the role dependency graph $G = (V, E)$ is defined as follows. The $V$ node set of $G$ is the inverse closure of the set of roles occurring in $\mathcal{A}$ and $\mathcal{R}$. There is a directed edge from $P_i$ to $P_j$ and from $\text{Inv}(P_i)$ to $\text{Inv}(P_j)$, if and only if $P_i \sqsubseteq P_j \in \mathcal{R}$.

Let $G$ be a role dependency graph wrt. an ABox $\mathcal{A}$ and an RBox $\mathcal{R}$ and let us consider its reduced graph $G_r$. Notice that a node of $G_r$ is a component of the original graph whose elements are equivalent roles. Also notice that if roles $R_1, \ldots, R_n$ are in one component then $\text{inv}(R_1), \ldots, \text{inv}(R_n)$ are in one component as well. There is an edge between components $C_1$ and $C_2$ if and only if all the roles in $C_1$ are subroles of the roles in $C_2$, according to an axiom in $\mathcal{R}$.

Let us nominate one of the roles in a component $E$ as the main equivalent of $E$, denoted by $\text{Main}(E)$. This role represents the whole equivalence group. We can select the main equivalent of a group in an arbitrary way: let us say that the $\text{Main}(E)$ is the role that comes first according to the standard Prolog ordering.

Finally, we build another graph, called the asymmetric inverse dependency graph. This graph has the same set of nodes as the reduced role dependency graph, but the edges correspond to an asymmetric inverse relation between the equivalence classes.

Definition 32. Let $G_r = (V, E)$ be the reduced role dependency graph wrt. an ABox $\mathcal{A}$ and an RBox $\mathcal{R}$. The asymmetric inverse dependency graph $I = (Q, W)$ is defined as follows. The node set of $I$ is the same as that of $G$, i.e. $Q = V$. Let $X, Y \in V$ be two components such that the inverse role of $\text{Main}(X)$ can be found in $Y$. $(X, Y) \in W$ if and only if the name of $\text{Main}(X)$ does not start with prefix $\text{inv\_}$.

According to the definition, the asymmetric inverse dependency graph is a bipartite graph where the nodes are partitioned into two non empty sets $V_1$ and $V_2$. An example of such graph is shown in Figure 2.26.

The idea behind the asymmetric inverse dependency graph is that now roles can be divided into two groups and during the transformation process we will only work with roles in one of these groups, let us say, roles in $V_1$. Any occurrence of a role that is in $V_2$ is replaced by a role from $V_1$. 

![Figure 2.26: An example for an asymmetric inverse dependency graph.](image-url)
**Definition 33.** Let \( G = (V, E) \) be an asymmetric inverse dependency graph with \( V_1 \) and \( V_2 \) being the two partitions. The *replacement* of a role invocation \( P \), denoted by \( \text{Rep}(P) \), wrt. \( G \) is the following. If \( P \in V_1 \), then we simply replace the name of \( P \) with \( \text{Main}(P) \). Otherwise, let \( C \) denote the equivalence group of the name of \( P \). We replace \( P \) with \( P' \), where the two arguments of \( P' \) are the same as those of \( P \), but in reverse order. The name of \( P' \) is \( \text{Main}(X) \), where \( X \) is the component in \( V_1 \) for which \( (X, C) \in E \) holds.

For example, \( \text{Rep}(\text{inv_hasChild}(A, B)) = \text{hasChild}(B, A) \). We can now define the transformation schemes for equivalent roles and role hierarchies.

**Transforming equivalence groups** The transformation scheme corresponding to a node of \( G_r \) which contains roles \( R_1, \ldots, R_n \) is summarised in Figure 2.27. Here \( R_1 \) is the main equivalent: all the other roles \( R_2, \ldots, R_n \) are reduced to querying \( R_1 \). Querying \( R_1 \) is done directly from the ABox (this is denoted by the abox: prefixes).

\begin{verbatim}
R1(A, B) :- abox:R1(A, B).
R1(A, B) :- abox:inv_R1(B, A).
R1(A, B) :- abox:R2(A, B).
R1(A, B) :- abox:inv_R2(B, A).
... 
R1(A, B) :- abox:Rn(A, B).
R1(A, B) :- abox:inv_Rn(B, A).
...
R2(A, B) :- R1(A, B).
...
Rn(A, B) :- R1(A, B).
\end{verbatim}

Figure 2.27: Transformation scheme for equivalent roles \( R_1, \ldots, R_n \).

Note that this scheme works also for symmetric roles, i.e. roles which are equivalent to their inverses. If roles \( \text{hasChild}/2 \) and \( \text{inv_hasChild}/2 \), for example, are in one equivalence group the transformation scheme yields the following code:

\begin{verbatim}
hasChild(A, B) :- abox:hasChild(A, B).
hasChild(A, B) :- abox:inv_hasChild(B, A).
hasChild(A, B) :- abox:inv_hasChild(A, B).
hasChild(A, B) :- abox:hasChild(B, A).
inv_hasChild(A, B) :- hasChild(A, B).
\end{verbatim}

**Transforming role hierarchies** Now we turn our attention to transforming role hierarchies. First note that because of the way \( G_r \) is built, if \( (C, D) \in E \), then \( (\text{Inv}(C), \text{Inv}(D)) \in E \) as well. The transformation of one of these axioms, however, is unnecessary.

Now, the idea is that we always populate the main equivalents from the main equivalents: namely, if there is an edge between nodes \( C \) and \( D \) in the reduced role dependency graph such that \( C \) is in \( V_1 \), then we generate the following piece of program code

\begin{verbatim}
\text{Rep}(D(A, B)) :- S(A, B).
\end{verbatim}

, where \( S = \text{Main}(C) \). Note that because role \( \text{Rep}(D(A, B)) \) is a main equivalent, all the other roles in the same equivalence group will also be populated from \( S/2 \) (cf. the transformation scheme for equivalent groups). Also note that \( S/2 \) may also have equivalents, but because \( S/2 \) is the main equivalent it captures all the individuals covered by the equivalents of \( S/2 \) (cf. the transformation scheme for equivalent groups).
To sum up, we have managed to transform the role axioms of a $SHIQ$ knowledge base into Prolog code that does not require loop elimination to terminate.

2.5 The DLog system

In this section we first introduce the software architecture of the DLog system. Next, we discuss the implementation specific optimisations we have developed. Finally, we present the various options one can use when invoking the DLog system.

2.5.1 Architecture

DLog is a resolution based Description Logic ABox reasoner over the $SHIQ$ DL language which implements the techniques described in the chapter. DLog has been developed in SICStus Prolog, involving a total of approximately 182KB of Prolog source code.

The general architecture of the DLog system is shown in Figure 2.28. The system can be used as a server or as a standalone application. In either case, the input of the reasoning process is provided by a DIG file. DIG [9] provides a standardised interface for Description Logic Reasoners that uses XML syntax.

![Figure 2.28: Architecture of the DLog system.](image)

The input file has three parts: the (potentially) large ABox, the smaller TBox and the Ask part describing the queries. The content of the ABox is asserted into module abox, either without modifications or (if we apply indexing) together with the index predicates. Note that the ABox can also be supplied as a database. This is essential for really large data sets.

The content of the TBox is transformed into a Prolog program according to the techniques described in previous sections. This program is then compiled into module tbox.

The content of the Ask part in the DIG input contains the user queries. In the simplest case the user poses an instance retrieval query which directly corresponds to a concept in the TBox. Such cases are answered by directly invoking the appropriate choice predicate. In the more complex case we have a conjunctive query as introduced in Section 2.3.3.

We handle conjunctive queries by reducing the problem of query answering to a normal DL reasoning task [58]. We simply apply body reordering (Section 2.4.4) and decomposition (Section 2.4.7) on a conjunctive query and use the normal Prolog execution for the resulting goal. We realise that much more sophisticated techniques are available [99], but at the moment our approach seems to be efficient enough.
2.5.2 Low-level optimisations

During the implementation we have applied several optimisations that can be considered as implementation specific or low level. Below we give a brief summary of these optimisations.

**Loop and ancestor separation** It is worth separating the data structures used for loop elimination and ancestor resolution. This way we can update them separately resulting in more efficient execution.

**Hashing** Instead of simple lists it is worth using a more efficient data structure to store goals used for loop elimination and ancestor resolution. For this purpose we use a special hashing library written in Prolog and C using the SICStus Prolog foreign interface. As an example for the usefulness of hashing consider the following DL knowledge base (another example is the Iocaste knowledge base which will be evaluated in detail in the next section).

\[
\begin{align*}
\exists \text{hasFriend}\cdot \neg \text{Alcoholic} \\
\exists \text{hasParent}\cdot \neg \text{Alcoholic} \\
\text{hasParent}(i_1, i_2) \cdot \text{hasParent}(i_1, i_3) \cdot \text{hasFriend}(i_2, i_3).
\end{align*}
\]

Here, in the TBox, we have described that if someone has a friend who is alcoholic then she is not alcoholic (she sees a bad example). We also described that if someone has a non-alcoholic parent then she is not alcoholic either (she sees a good example). The ABox contains two \text{hasParent} and one \text{hasFriend} relation, and nothing about someone being alcoholic or non-alcoholic. Interestingly, it is possible to conclude that \(i_1\) is non-alcoholic as one of her parents is surely non-alcoholic (as at least one out of two people who are friends has to be non-alcoholic).

For certain ABoxes, the Prolog translation of this knowledge base runs quadratically in the number of \text{hasParent} relations. Such an ABox is the following:

\[
\begin{align*}
\text{hasParent}(i_k, i_{k+1}), & \quad k = 1, \ldots, n \\
\text{hasFriend}(i_{n+1}, i_{n+2}) \\
\text{inv}\_\text{hasParent}(i_{n+1+t}, i_{n+2+t}), & \quad t = 1, \ldots, n
\end{align*}
\]

Here for every individual \(i_{n+1+t}, t > 0\) we have to check if the ancestor list contains the term \text{not}\_\text{Alcoholic}(i_{n+1+t}), which has linear cost wrt. the size of the ancestor list. This is assuming that the check for a given ancestor is performed by a linear scan of the ancestor list. The quadratic time complexity can be reduced to (nearly) linear when a hash table is used for storing the ancestors which results in (nearly) constant time ancestor check.

**Place of the update operations** As updating a hash structure is considered to be more expensive than adding a new element to an existing list (cf. line 7 in Figure 2.25), it is worth updating these structures only right before they are used (i.e. they should not always be updated at the very beginning of a body).

**Clause-level categorisation** In addition to predicate categorisation (see Section 2.4.3) it is worth categorising the individual clauses of a predicate \(P\) as well. For example, even if a predicate is recursive, some of its clauses will never lead to recursive calls of this predicate. For these clauses, there is no point updating the loop data structure. Similarly, if from a clause of \(P\) we cannot reach \text{not}\_P, we do not need to update the ancestor data structure.

2.5.3 Execution parameters

In DLog most of the optimisations discussed in Section 2.4 can be enabled/or disabled, resulting in different generated Prolog programs. The possible parameter settings are summarised below (in parentheses we give the parameter values allowed, the first value is the default):
decompose(yes/no): whether to decompose the bodies (Section 2.4.7)
indexing(yes/no): whether to generate index predicates for roles (Section 2.4.5)
projection(yes/no): whether to calculate supersets (Section 2.4.8)
filtering(yes/no): whether to do filtering (Section 2.4.2)
ground_optim(yes/no): whether to use ground goal optimisation (Section 2.4.6)
orphan(first/general): whether orphan calls are brought to the beginning of the clause or handled like general concept calls
hashing (yes/no): whether to apply hash tables or lists for storing ancestors

2.6 Performance Evaluation

This section presents a comparison of the performance of DLog with that of existing DL reasoning systems. The aim is to obtain an insight of the practical applicability of the ideas described in Sections 2.3 and 2.4.

During the tests we have also found several anomalies which resulted in significant performance drops in the case of certain DL reasoners. We believe that most of these will be fixed by the respective authors in the near future. Here, however, we took each system “as it is”, which means that we examined how their most up-to-date version performs on various inputs.

Our tests suggest that resolution-based techniques are very promising in practical applications, where the TBox is relatively small, but the ABox is huge.

2.6.1 Test environment

We have compared our system with three state-of-the-art description logic reasoners: RacerPro 1.9.0, Pellet 1.5.0 and the latest version of KAON2 (august 2007). RacerPro [50] is a commercial system, Pellet [122] is open-source, while KAON2 [99] is free of charge for universities for non-commercial academic usage. We did not consider the other available reasoning systems mainly because they are either outdated or they do not support ABox reasoning at all (e.g. this is the case for the widely used FaCT system).

We contacted the authors of each reasoning system in order to know the preferred sequence of API calls for running our tests. From one of them we did not receive any response so we used the API according to the documentation. The benchmarks were executed by a test framework we have specifically developed for testing DLog and other systems. For each query, we started a clean instance of the reasoner and loaded the test knowledge base. After this, we measured the time required to execute the given query. Each query was repeated 5 times. The best and the worst results were excluded and the remaining three were used for calculating an average. In case the execution was very quick (less than 10 milliseconds) we have repeated the test 1000 times and calculated the average. We made sure that all systems return the same answers for the given queries.

The tests were performed on a Fujitsu-Siemens S7020 laptop with a Pentium-M 1.7GHz processor, 1.25GB memory, Ubuntu Linux 7.04 with Linux kernel 2.6.20-16 and SICStus Prolog 3.12.8. The version of the Java virtual machine, used for KAON2 and Pellet, is 1.5.0.

2.6.2 Test Ontologies

For the benchmark we have used three family of ontologies. The first one corresponds to the Locaste problem introduced in Figure 2.1. For performing this test we created a program that is able to generate random Locaste ontologies based on certain initial parameters (number of nodes, branching factor, etc.).

First we used this program to generate “clean” Locaste ontologies, i.e. DL knowledge bases with ABox containing nothing else than given sized Locaste patterns (cf. Figure 2.2). These ontologies are named cN, where N is the size of the pattern. For example, c100 denotes the DL knowledge base with one TBox axiom and an ABox containing 102 individuals according to Figure 2.2 with n = 100.

We have also generated “noisy” Locaste knowledge bases. By “noise” we mean irrelevant individuals, role and concept assertions which we added to the ABoxes (for example pairs in hasChild relation that
are irrelevant from the point of execution). We did this in order to be able to measure how sensitive the inference engines are to this kind of ABox modification. By using irrelevant nodes we actually simulate real life situations, because during a database search it is rarely the case that we look for the whole content. The noisy Iocaste knowledge bases are named \( n_1, n_2, n_3 \) and \( n_4 \). The properties of the clean and noisy Iocaste ontologies are summarised at the end of the chapter in Table 2.8.

Here we can see that the largest clean ontology contains a bit more than 20000 ABox axioms, while the largest noisy ontology has more than 30000 axioms. Every ontology contains only a single TBox axiom (cf. Figure 2.1). In the table we also give the size of the corresponding DIG files (in megabytes) and the time it took for the system to parse these files and create the intermediate knowledge base (load time).

The other sections in Table 2.8 correspond to the various parameter settings we have tried the DLog system with. For every setting we give the translation time (the time it took to generate the Prolog program according to Section 2.4) and the time it took the SICStus Prolog system to actually compile the generated program. The total time is the sum of three values: the load time, the translation time and the compile time.

For testing from the possible \( 2^8 \) option variations (cf. Section 2.5.3) we use only the following ones:

- \( \text{base} \): everything is left as default
- \( \{ g(n) \} \): do not use ground goal optimisation
- \( \{ p(n) \} \): do not use projection
- \( \{ f(n) \} \): do not use filtering
- \( \{ i(n) \} \): do not use indexing
- \( \{ o(n) \} \): consider orphan goals as normal predicates
- \( \{ d(n) \} \): do not decompose the bodies
- \( \{ p(n) \} + \{ d(n) \} \): do not use projection and decomposition
- \( \{ o(n) \} + \{ d(n) \} \): do not use decomposition and consider orphan goals normal

We can conclude that every settings have more or less the same compile-time properties. However, in the setting \( \{ i(n) \} \) we do not have to generate index predicates, which results in a more compact code. This means shorter translation and compilation times. Note that the Iocaste ontologies, both the clean and the noisy ones, use the \( \mathcal{ALC} \) DL language.

The second ontology we used for testing is called VICODI [105]. VICODI is a manually created ontology about European history, the result of an EU-IST programme. Technically VICODI is an \( \mathcal{ALCH}i \) ontology with a fairly simple TBox and a huge ABox. We have obtained VICODI from the VICODI homepage as a Protege project. This we have exported into OWL and DIG formats using Protege 3.3.1 and used these as inputs for the various reasoners. The sizes of these files were 9.5 and 23 megabytes, respectively. The VICODI TBox consists of 182 concept and 9 role subsumption axioms. The ABox contains 84550 role axioms and 29614 concept axioms.

Finally, we have also tested our system on the Leigh University Benchmark (LUBM). This ontology was developed by [48] specifically as a benchmark for testing performance of ontology reasoners. The ontology describes organisational structure of universities and it uses the \( \mathcal{ALCH}1 \) language. The ABox part can be automatically generated by specifying the number of universities.

In our tests we have used ontologies denoted by \( \text{lubm1}, \text{lubm2}, \text{lubm3} \) and \( \text{lubm4} \), where each ontology is generated automatically by specifying the number of universities. All LUBM ontologies contain 36 concept inclusion, 6 concept equivalence, 5 role inclusion and 4 role equivalence axioms. They also contain a transitive role and 21 domain and 18 range restrictions. The number of ABox axioms in the LUBM ontologies and their sizes in megabytes are summarised in Table 2.1.

### 2.6.3 Results

We now present the execution results for the Iocaste, VICODI and LUBM ontologies. For each case we give a detailed explanation of the results.
Table 2.1: The properties of the LUBM test ontologies.

<table>
<thead>
<tr>
<th></th>
<th>testfile</th>
<th>lubm1</th>
<th>lubm2</th>
<th>lubm3</th>
<th>lubm4</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWL filesize (MBytes)</td>
<td>6.90</td>
<td>15.84</td>
<td>23.24</td>
<td>32.97</td>
<td></td>
</tr>
<tr>
<td>DIG filesize (MBytes)</td>
<td>16.57</td>
<td>37.99</td>
<td>58.80</td>
<td>81.74</td>
<td></td>
</tr>
<tr>
<td>concept assertions</td>
<td>18128</td>
<td>40508</td>
<td>58897</td>
<td>83200</td>
<td></td>
</tr>
<tr>
<td>role assertions</td>
<td>49336</td>
<td>113463</td>
<td>166682</td>
<td>236514</td>
<td></td>
</tr>
</tbody>
</table>

The Oedipus family

The execution results of the DLog system on the Iocaste tests are summarised in Table 2.9. Here, for every parameter settings we show four values. Three values (loop, ancres, orphancres) are statistical information, describing the number of loop elimination, ancestor resolution and orphan ancestor resolution (ancestor resolutions that occurred in orphan goals). Finally, we show the most important value, the runtime.

With the best settings (base) DLog solved every task within a fraction of a second, including the biggest clean and the biggest noisy cases as well. Actually, using projection (cf. Section 2.4.8) seems to be a key factor as without it the performance dramatically drops. We can also conclude that the lack of multiple argument indexing (cf. Section 2.4.5) has very negative effect on the execution time. We can also see that with the last parameter setting DLog was not able to solve every tasks (denoted by -). In this setting we do not use decomposition and we treat orphans as normal predicates. The reason why this setting has the worst performance is that here the orphan goal not_Ans(D, B) is placed at the very end of the corresponding det_Patricide/2 clause (cf. Figure 2.19).

We have compared the results of DLog corresponding to the base parameter variation with that of the other three reasoning systems. These aggregate results are summarised in Table 2.2.

Table 2.2: Aggregate results for the Iocaste ontologies (times are given in seconds)

<table>
<thead>
<tr>
<th>Query</th>
<th>Ans</th>
<th>Testfile</th>
<th>c10</th>
<th>c20</th>
<th>c100</th>
<th>c1000</th>
<th>n1</th>
<th>n2</th>
<th>n3</th>
<th>n4</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLog</td>
<td></td>
<td>load</td>
<td>0.07</td>
<td>0.08</td>
<td>0.15</td>
<td>0.33</td>
<td>1.47</td>
<td>0.14</td>
<td>0.24</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td></td>
<td>runtime</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.11</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total</td>
<td>0.07</td>
<td>0.08</td>
<td>0.15</td>
<td>0.34</td>
<td>1.58</td>
<td>0.14</td>
<td>0.24</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>KAdON2</td>
<td>load</td>
<td>0.45</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.46</td>
<td>0.60</td>
<td>0.97</td>
<td>2.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>runtime</td>
<td>0.72</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.67</td>
<td>4.72</td>
<td>63.60</td>
<td>425.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total</td>
<td>1.17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1.13</td>
<td>5.32</td>
<td>64.57</td>
<td>427.53</td>
</tr>
<tr>
<td></td>
<td>RacerPro</td>
<td>load</td>
<td>0.01</td>
<td>0.01</td>
<td>0.03</td>
<td>0.51</td>
<td>4.68</td>
<td>0.03</td>
<td>0.10</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td></td>
<td>runtime</td>
<td>0.07</td>
<td>0.09</td>
<td>0.15</td>
<td>1.68</td>
<td>79.91</td>
<td>0.10</td>
<td>0.47</td>
<td>1.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total</td>
<td>0.08</td>
<td>0.10</td>
<td>0.18</td>
<td>2.19</td>
<td>84.59</td>
<td>0.13</td>
<td>0.57</td>
<td>2.44</td>
</tr>
<tr>
<td></td>
<td>Pellet</td>
<td>load</td>
<td>1.27</td>
<td>1.35</td>
<td>1.44</td>
<td>2.19</td>
<td>1.32</td>
<td>1.53</td>
<td>2.36</td>
<td>5.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>runtime</td>
<td>0.19</td>
<td>0.32</td>
<td>1.31</td>
<td>456.40</td>
<td>0.33</td>
<td>0.80</td>
<td>2.48</td>
<td>23.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total</td>
<td>1.46</td>
<td>1.68</td>
<td>2.76</td>
<td>458.58</td>
<td>1.65</td>
<td>2.33</td>
<td>4.84</td>
<td>29.87</td>
</tr>
</tbody>
</table>

Here, for each Iocaste ontology and each reasoning system we give the following values: the load time, the runtime and their sum, the total time. The load time in the case of the DLog system includes parsing, translating and SICStus compilation (cf. Table 2.8). In the case of the other systems, load time is the time it takes to reach the point when a query can be posed (we do not have detailed information what the systems are actually doing here other than parsing the input). Note that physically DLog is given a bigger input than the other systems as the DIG format is more verbose than the OWL. Throughout this chapter, whenever we compare various systems/options, the best total time is given in bold.
KAON2 showed a very poor performance on the clean Iocaste ontologies: c10 was the only test case it was able to solve within the time limit. To understand better what is going on we have tested KAON2 with clean Iocaste patterns of length $n = 11, \ldots, 15$. The results of this experiment are summarised in Table 2.3. Here we can see that KAON2 scales very badly when increasing the size of the pattern. Note that the difference between the consecutive test files are absolutely minimal. For example, ontology c15 contains only one instance more (and two role axioms more) than ontology c14.

<table>
<thead>
<tr>
<th>test file</th>
<th>c10</th>
<th>c11</th>
<th>c12</th>
<th>c13</th>
<th>c14</th>
<th>c15</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime</td>
<td>0.72</td>
<td>0.68</td>
<td>3.51</td>
<td>16.18</td>
<td>17.03</td>
<td>309.91</td>
</tr>
</tbody>
</table>

Table 2.3: Performance of KAON2 on the Iocaste ontologies (times are in seconds).

Another interesting thing is that KAON2 actually ran faster on ontology c11 than c10. It also seems to scale reasonably well (at least comparing to the other cases) from c13 to c14.

In the case of the noisy ontologies KAON2 also behaved strangely. Although it was able to solve all the tests within 10 minutes, we definitely expected KAON2 to solve these cases much quicker as, similarly to DLog, KAON2 uses resolution, which means that it should be resistant to noise to a large extent.

We have actually learnt [100] that in KAON2 many things depend on the order of rule applications, something which is a very difficult task to set properly. Choosing a bad order may result in a big performance drop. This can be a reason for the anomalies we have seen in the case of the Iocaste ontologies.

RacerPro was able to solve every test case within the time limit. It showed a very consistent behaviour both in the case of the clean and the noisy variants. From the test results it seems that RacerPro scales linearly although with a much worse constant than DLog. As a tableau based reasoner, RacerPro showed a surprisingly good performance in the case of the largest noise variant n4, as well (23.25 seconds).

Pellet was nearly as fast as RacerPro in the case of the noisy variants. On the clean Iocaste ontologies, however, it was clearly outclassed by RacerPro as Pellet was not able to solve c10000 within 10 minutes and in all the other cases it was fairly slow as well. We have also found that several cases Pellet threw certain Java exceptions on the very same input it accepted before and after as well. We guess that this can be because of the use of Java hashcodes.

As a conclusion we can say that considering the runtimes as well as the total times in the case of the Iocaste ontologies, DLog is faster by several orders of magnitude than the other existing ABox reasoners.

**Using databases** To eliminate creating large internal Prolog databases only to store the ABox, we can actually put the content of the ABox into a real database and use DLog to generate a program from only the TBox. We have used this technique for the largest noisy variant n4 with options [i(n)]. Here, according to Table 2.8 and Table 2.9, we use 1.41 seconds for compile time work and 0.02 seconds for runtime. By using a database for storing the content of the ABox, we expect drastic decrease in the total compile time, while slight increase in the execution time.

The actual (MySQL) database contains 15 tables, from which 10 correspond to concepts (i.e. they have only on column), while the rest corresponds to roles (i.e. they have two columns). Note that because of the top-down execution we apply the Prolog program generated from the TBox actually accesses only tables Patricide, not_Patricide and hasChild. We have 5058 pairs in hasChild relation, 855 instances are known to be patricide and 314 are known to be not patricide.

The execution results are summarised in Table 2.4. The database variant of n4 enumerated in 0.36 seconds all the instances of concept Ans. This, compared to the original 0.02 seconds is much slower. However, the time we spent at compile-time was altogether 0.07 seconds, resulting in a total execution time of 0.43 seconds.

We note that although it may seem like using database for storing the ABox (that would fit into memory anyway) is beneficial because of the reduced compile-time work only, this is not necessarily so. We believe that in the case of large data sets and complex queries (e.g. executing query predicates) execution time can also be better than that of the in-memory variant.

As a conclusion we can state that using database to store the content of an ABox is feasible and in the case of the Iocaste ontologies it provides better overall performance than normally.
Table 2.4: Comparing the in-memory and database version of the n4 test.

<table>
<thead>
<tr>
<th></th>
<th>DLog</th>
<th>load</th>
<th>translate</th>
<th>compile</th>
<th>runtime</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-memory</td>
<td>0.88</td>
<td>0.52</td>
<td>0.01</td>
<td>0.02</td>
<td>1.43</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>0.05</td>
<td>0.01</td>
<td>0.01</td>
<td>0.36</td>
<td>0.43</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.5: The effect of hashing on the Locaste ontologies.

We can see that in the case of the large Locaste patterns (c100 and c10000) the hashing implementation significantly outperforms the simple list solution.

### VICODI

To test the performance of the DL reasoners on the VICODI ontology, we used the following two queries borrowed from [99]:

\[
\begin{align*}
VQ_1(X) & \equiv \text{Individual}(X) \\
VQ_2(X,Y,Z) & \equiv \text{Military-Person}(X), \text{hasRole}(Y,X), \text{related}(X,Z)
\end{align*}
\]

The results are summarised in Table 2.6. The DLog system used 8.61 seconds to load the VICODI ontology. From this, 4.91 seconds were actually spent on parsing the input and transforming the DL knowledge base into DL predicates. DLog used 3.38 seconds to generate the Prolog code. The rest (0.36 seconds) was used by SICStus Prolog to compile the generated Prolog program. After this, the execution was nearly instantaneous: 0.05 seconds for \(VQ_1\) and 0.09 seconds for \(VQ_2\).

<table>
<thead>
<tr>
<th></th>
<th>DLog</th>
<th>KAON2</th>
<th>RacerPro</th>
<th>Pellet</th>
</tr>
</thead>
<tbody>
<tr>
<td>load time</td>
<td>8.61</td>
<td>5.88</td>
<td>34.96</td>
<td>-</td>
</tr>
<tr>
<td>runtime</td>
<td>0.05</td>
<td>0.36</td>
<td>76.48</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>8.66</td>
<td>6.24</td>
<td>111.44</td>
<td>-</td>
</tr>
<tr>
<td>runtime</td>
<td>0.09</td>
<td>0.35</td>
<td>76.61</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>8.70</td>
<td>6.23</td>
<td>111.57</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.6: Aggregate results for the VICODI ontology.

RacerPro spent nearly 35 seconds for loading the ontology. The execution of \(VQ_1\) was fairly slow: it took 76.48 seconds to enumerate all the instances of class \text{Individual}. We also measured the execution time by first checking the consistency of the ABox, then preparing the query answering engine before posing the query itself. Consistency check took 65.86 seconds, query engine preparation 1.29 seconds and the query itself 8.25 seconds. This results in a total time of 75.40, which (as expected) matches the total time of simply loading and querying.
In the case of VQ2, RacerPro produced nearly the same results. We believe this is because it spends most of its time in checking ABox consistency, which requires the same amount of time in both queries.

Pellet was unable to answer any of the queries within the 10 minutes time limit. We believe that Pellet properly read the input as we could formulate VICODI queries that Pellet was able to answer, but this was not the case for queries VQ1 and VQ2. We have also tried the Windows version of Pellet, but we have experienced the same behaviour. Actually, in [99] Pellet 1.3 beta was tested against the VICODI ontology with acceptable results. Thus it seems that recent changes in the Pellet are responsible for the performance drop we have found.

KAON2 could not read the VICODI owl input we generated with Protege: we got an exception. To be able to run the tests, we used a version of the ontology specifically made for KAON2 (available on the VICODI website). This version of the ontology is physically twice as large as the normal OWL dialect (i.e. it is 18MB). On this, KAON2 was very convincing. It took 5.88 seconds to load the ontology and 0.36 seconds to answer query VQ1. Answering query VQ2 was even a bit faster, it required 0.35 seconds. We note that this format of the VICODI ontology is not supported by neither RacerPro, nor Pellet, so the comparison is not fully fair.

As a conclusion we can say that KAON2 had the best overall performance when dealing with the VICODI ontology. DLog answered the queries even faster than KAON2, but for the compile-time tasks we needed a few seconds more. This prevented DLog to be the winner here. We note, however, that the DIG input is larger by 5MB than the KAON2 version of the VICODI ontology which naturally results in more compile-time work for us.

**LUBM**

We have tested the LUBM ontologies with the following two queries:

| LQ1(X)  | ≡ Person(X), hasAlumnus(http://www.University0.edu, X) |
| LQ2(X,Y) | ≡ Chair(X), Department(Y), worksFor(X, Y), subOrganizationOf(Y, http://www.University0.edu) |

These queries were selected from the 14 test queries available on the LUBM homepage. Answering LQ1 requires proper handling of role hierarchies and inverses. LQ2 is interesting as it is a complex conjunctive query. The execution results are summarised in Table 2.7. For DLog we used the base parameter setting, i.e. we apply every optimisations we can.

Loading lubm1 took DLog 6.96 seconds. From this it took 5.29 seconds to read the DIG file and create the DL predicates. We needed 1.47 seconds to generate the Prolog code. Finally, it took 0.18 seconds for SICStus Prolog to compile the generated code. Answering LQ1 required only 0.26 seconds, while LQ2 was answered instantaneously.

Although loading the larger lubm ontologies required much more time, answering LQ1 was only a bit slower in the largest case than in the case of lubm1. The other query, LQ2, was executed instantaneously on all of the LUBM ontologies.

Note that for DLog a significant part of the compile-time work is the generation of the index predicates (cf. Section 2.4.5). This effectively doubles the number of the role assertions. Using this optimisation becomes unnecessary if we use a Prolog system with multiple argument indexing or we store the ABox externally in a database - which is the intended use of the DLog system. Also note that the DIG input given to DLog is significantly larger (cf. Table 2.1) than the OWL input the other reasoning systems use.

KAON2 behaved very nicely on the LUBM ontologies: it was able to answer both queries LQ1 and LQ2 on all ontologies very quickly. We note that actually the official version of KAON2 was unable to solve the LUBM tests due to certain technical problems. After contacting the author, these issues were quickly fixed, however.

RacerPro managed to solve both queries on ontologies lubm1 and lubm2 with total times between 56.94 and 211.35 seconds. Here we can see the usual pattern: there is no real difference between the execution times of LQ1 and LQ2. Unfortunately, on the bigger ontologies, RacerPro had memory problems.

Pellet solved the queries only on the smallest LUBM ontology. This required 48.69 and 48.79 seconds. On the larger ontologies Pellet did not signal memory problems, but simply ran out of the 10 minutes limit.
Table 2.7: Aggregate results for the LUBM ontologies

Note, than in the case of RacerPro and Pellet we also show the setup time which is the time of the ABox consistency tests these systems always do at startup. We can see that RacerPro really spends most of its time in this phase. On the other hand, Pellet spends fairly little on consistency checking.

To conclude the LUBM tests we can say that DLog and KAON2 were the only systems able to solve all the queries on all the LUBM ontologies. Out of these two systems DLog emerges as the winner by a small margin (although in terms of runtime DLog is much faster). It is again worth noticing that, as in other cases, the execution times of DLog and KAON2 are very good compared to that of the Tableau-based description logic reasoners.

2.7 Future work

We now give a brief overview of future work on the DLog system, for improving its performance as well as extending its capabilities.

**Partial evaluation**  Recall property (p1) in Definition 1, which states that each DL clause either contains a binary literal or it contains at most one variable. Note that the body of the latter type of clauses is actually a conjunction of concept goals. It is because of such clauses that the ancestor list can be non-ground.

One can apply partial evaluation techniques, such as in [137], to unfold clauses containing no binary literals. Such unfolding should be continued until each clause contains either a binary literal or a unary literal corresponding to an ABox predicate. Both such types of literals ensure that all their arguments are ground upon exit. This means that we no longer need to cater for executing unary predicates with uninstanitated arguments (except for the outermost query predicate). Also, the ancestor list becomes ground, which simplifies hashing. The absence of logic variables in the data structures opens up the possibility of compiling into Mercury code, rather than Prolog, which is expected to execute much faster than standard Prolog.

**Tabling in the presence of ancestors**  It is often the case that the same goal is invoked multiple times during query execution. To prevent unnecessary execution of such goals tabling [138] can be used. Note, however, that these goals are invoked in different contexts, i.e. with different ancestor list arguments, in
which case traditional tabling is of no use. A specialised form of tabling can be developed for DLog execution, which keeps track of those ancestors that are actually required for the successful completion of a given goal invocation. This is expected to improve the execution of queries on knowledge bases heavily relying on ancestor resolution, such as the Alcoholic example of Section 2.5.2.

**Further improvements** As explained in Section 2.5.1, presently we apply a simple query ordering technique for execution of conjunctive queries. This can be improved using the techniques of [99]. Furthermore, we presently do not use statistical information in query ordering. Techniques relying on statistical data are well researched in the context of databases. The use of such techniques in DLog should be investigated as these can result in significant increase of execution performance.

The external interfaces of DLog need to be extended to support new input formats, in addition to the DIG standard. We presently have an experimental interface to support database queries. Further work is needed to implement general interfaces to database systems, including optimisations such as passing appropriate query conjunctions to database management systems, instead of single queries.

### 2.8 Summary and conclusions

In this chapter we have presented the description logic reasoning system DLog. Unlike the traditional Tableau based approach, DLog determines the instances of a given $\mathcal{SHIQ}$ concept by transforming the knowledge base into a Prolog program. This technique allows us to use top-down query execution and to store the content of the ABox externally in a database, something which is essential when large amounts of data is involved.

We have compared DLog with the best available ABox reasoning systems. From the test results we can conclude that in all of the scenarios DLog is significantly faster than traditional reasoning systems. In most of the cases DLog also outperforms KAON2 which uses a similar resolution based idea as DLog.

We note that trends and behaviours of the various algorithms on certain inputs can be more interesting than the actual runtimes (as this can be very much affected by specific implementation details). Considering also this, we argue that DLog and KAON2 are much better suited for large data sets than tableau-based description logic reasoners.

As an overall conclusion, we believe that our results are very promising and clearly show that description logic is an interesting application field for Prolog and logic programming.
<table>
<thead>
<tr>
<th>Testfile</th>
<th>c10</th>
<th>c20</th>
<th>c100</th>
<th>c1000</th>
<th>c10000</th>
<th>n1</th>
<th>n2</th>
<th>n3</th>
<th>n4</th>
</tr>
</thead>
<tbody>
<tr>
<td>size (MB)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.19</td>
<td>1.88</td>
<td>0.01</td>
<td>0.06</td>
<td>0.35</td>
<td>2.82</td>
</tr>
<tr>
<td>axioms (T)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>axioms (A)</td>
<td>22</td>
<td>42</td>
<td>202</td>
<td>2002</td>
<td>20002</td>
<td>100</td>
<td>646</td>
<td>3897</td>
<td>30797</td>
</tr>
<tr>
<td>load</td>
<td>0.04</td>
<td>0.06</td>
<td>0.13</td>
<td>0.23</td>
<td>0.78</td>
<td>0.12</td>
<td>0.21</td>
<td>0.23</td>
<td>0.88</td>
</tr>
<tr>
<td>translate</td>
<td>0.00</td>
<td>0.01</td>
<td>0.1</td>
<td>0.08</td>
<td>0.67</td>
<td>0.01</td>
<td>0.02</td>
<td>0.13</td>
<td>1.10</td>
</tr>
<tr>
<td>compile</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>total</td>
<td>0.07</td>
<td>0.08</td>
<td>0.15</td>
<td>0.33</td>
<td>1.47</td>
<td>0.14</td>
<td>0.24</td>
<td>0.38</td>
<td>1.99</td>
</tr>
<tr>
<td>g(n)</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.08</td>
<td>0.69</td>
<td>0.00</td>
<td>0.04</td>
<td>0.18</td>
<td>1.04</td>
</tr>
<tr>
<td>translate</td>
<td>0.00</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>compile</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>total</td>
<td>0.05</td>
<td>0.08</td>
<td>0.16</td>
<td>0.32</td>
<td>1.48</td>
<td>0.14</td>
<td>0.27</td>
<td>0.43</td>
<td>1.94</td>
</tr>
<tr>
<td>f(n)</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.07</td>
<td>0.72</td>
<td>0.01</td>
<td>0.02</td>
<td>0.16</td>
<td>1.05</td>
</tr>
<tr>
<td>translate</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>compile</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>total</td>
<td>0.06</td>
<td>0.08</td>
<td>0.16</td>
<td>0.33</td>
<td>1.50</td>
<td>0.14</td>
<td>0.25</td>
<td>0.39</td>
<td>2.19</td>
</tr>
<tr>
<td>i(n)</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.30</td>
<td>0.01</td>
<td>0.04</td>
<td>0.06</td>
<td>0.52</td>
</tr>
<tr>
<td>translate</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>compile</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>total</td>
<td>0.05</td>
<td>0.08</td>
<td>0.15</td>
<td>0.27</td>
<td>1.10</td>
<td>0.14</td>
<td>0.26</td>
<td>0.29</td>
<td>1.41</td>
</tr>
<tr>
<td>o(n)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.08</td>
<td>0.70</td>
<td>0.01</td>
<td>0.04</td>
<td>0.11</td>
<td>1.10</td>
</tr>
<tr>
<td>translate</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.03</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>compile</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>total</td>
<td>0.05</td>
<td>0.07</td>
<td>0.15</td>
<td>0.31</td>
<td>1.51</td>
<td>0.14</td>
<td>0.26</td>
<td>0.36</td>
<td>1.99</td>
</tr>
<tr>
<td>d(n)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
<td>0.08</td>
<td>0.71</td>
<td>0.01</td>
<td>0.02</td>
<td>0.15</td>
<td>1.05</td>
</tr>
<tr>
<td>translate</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>compile</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>total</td>
<td>0.04</td>
<td>0.07</td>
<td>0.15</td>
<td>0.32</td>
<td>1.50</td>
<td>0.14</td>
<td>0.25</td>
<td>0.39</td>
<td>1.95</td>
</tr>
<tr>
<td>p(n)+d(n)</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.07</td>
<td>0.71</td>
<td>0.01</td>
<td>0.02</td>
<td>0.12</td>
<td>1.06</td>
</tr>
<tr>
<td>translate</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>compile</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>total</td>
<td>0.06</td>
<td>0.09</td>
<td>0.17</td>
<td>0.32</td>
<td>1.51</td>
<td>0.14</td>
<td>0.24</td>
<td>0.37</td>
<td>1.95</td>
</tr>
<tr>
<td>o(n)+d(n)</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.08</td>
<td>0.73</td>
<td>0.02</td>
<td>0.03</td>
<td>0.12</td>
<td>1.04</td>
</tr>
<tr>
<td>translate</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>compile</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>total</td>
<td>0.06</td>
<td>0.08</td>
<td>0.15</td>
<td>0.33</td>
<td>1.53</td>
<td>0.16</td>
<td>0.25</td>
<td>0.36</td>
<td>1.96</td>
</tr>
</tbody>
</table>

Table 2.8: Properties of the test files (the times are given in seconds)
<table>
<thead>
<tr>
<th>Query</th>
<th>Testfile</th>
<th>Ans</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>c10</td>
<td>c20</td>
</tr>
<tr>
<td></td>
<td>solnum</td>
<td></td>
</tr>
<tr>
<td>base</td>
<td></td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>g(n)</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p(n)</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>f(n)</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i(n)</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>o(n)</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d(n)</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>p(n)+d(n)</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>o(n)+d(n)</td>
<td>loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>runtime</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 2.9: Results of the performance analysis
Chapter 3

DL-based conceptual modelling in information integration

This chapter presents an information integration system called SINTAGMA which supports the semantic integration of heterogeneous information sources using a metadata driven approach. The main idea of SINTAGMA is to build a so-called Model Warehouse, containing several layers of integrated models connected by mappings. At the bottom of this hierarchy there are the models representing the actual information sources. Higher level models represent virtual databases which can be queried, as the mappings provide a precise description of how to populate these virtual sources using the concrete ones. The implementation of SINTAGMA uses constraints and logic programming, for example, the complex queries are translated into Prolog goals.

This chapter focuses on my contribution in SINTAGMA allowing the information expert to use Description Logic (DL) based ontologies in the development of high abstraction level conceptual models. Querying these models is performed using the Closed World Assumption as we argue that traditional DL reasoning is less appropriate in the context of database oriented information integration environments.

3.1 Introduction

SINTAGMA is based on the SILK tool-set, developed within the EU FP5 project SILK (System Integration via Logic & Knowledge) [11]. SILK is a Prolog based, data centred, monolithic information integration system supporting semi-automatic integration on relational and semi-structured sources.

The SINTAGMA system extends the original framework in several directions. As opposed to the monolithic SILK structure, SINTAGMA is built from loosely coupled distributed components. The functionality has become richer as, among others, the system now deals with Web Services as information sources. The present paper discusses a recent extension of the system which allows the integration expert to use Description Logic models in the integration process.

The chapter is structured as follows. Section 3.2 gives a general introduction to the SINTAGMA system, describing the main components, the SILan modelling language, and the query execution mechanism. In the next section we discuss the description logic extension of SILan: we introduce the syntactic constructs and the modelling methodology. Section 3.4 describes the execution mechanism used when querying Description Logic models. Section 3.5 presents a fairly complex example, demonstrating the tools and techniques we have discussed so far. In Section 3.6 we examine related work. Finally, we conclude with a summary of our results.

The examples we use in the upcoming discussions are part of the integration scenario described in detail in Section 3.5. This scenario represents a world where we attempt to integrate various information sources about writers, painters and their work (i.e. books, paintings, etc.) and present this information in the form of abstract views.
3.2 SINTAGMA System Architecture

The overall architecture of the SINTAGMA system can be seen in Figure 3.1. The main idea of the system is to collect and manage meta-information on the sources to be integrated. These pieces of information are stored in the Model Warehouse, in the form of UML-like models [37], constraints and mappings. This way we can represent structural as well as non-structural information, such as class invariants, etc. The Model Warehouse resides in and is handled by the Model Manager component.

![Figure 3.1: The architecture of the SINTAGMA system](image)

We use the term mediation to refer to the process of querying SINTAGMA models. Mediation decomposes complex integrated queries to simple queries answerable by individual information sources, and, having obtained data from these, composes the results into a integrated form. Mediation is the task of the Mediator component.

Access to heterogeneous information sources is supported by wrappers. Wrappers hide the syntactic differences between the sources of different kinds, by presenting them to upper layers uniformly, as UML models. These models (called interface models) are entered into the Model Warehouse automatically. The following subsections give a brief description of the main SINTAGMA components.

3.2.1 The Model Manager

The Model Manager is responsible for managing the Model Warehouse (MW) and providing integration support, such as model comparison and verification (not covered in this paper). Here we focus on the role of the Model Warehouse.

The content of the MW is given in the language called SILan which is based on UML [37] and Description Logics [59]. The syntax of SILAN resembles IDL, the Interface Description Language of CORBA [111]. We demonstrate the knowledge representation facilities of SINTAGMA by a simple SILan example showing the relevant features of the meta-data repository (Figure 3.2).

The example describes the model Art containing two classes, Artist and Work. It also contains an association hasWork between artists and their works. We will explain the details of this example below.
Figure 3.2: SILan representation of the model Art

Semantics of SILan models

The central elements of SILan models are classes and associations, since these are the carriers of information. A class denotes a set of entities called the instances of the class. Similarly, an $n$-ary association denotes a set of $n$-ary tuples of class instances called links.

Classes can have attributes which are defined as functions mapping the class to a subset of values allowed by the type of the attribute. Classes can inherit from other classes. All instances of the descendant class are instances of the ancestor class, as well. In our example both Artist and Work inherit from the built-in class BuiltIns::DLAny (cf. lines 2 and 8). See Section 3.3.3 for more details.

Associations have connections, an $n$-ary association has $n$ connections. In an association some of the connections can be named, providing intuitive navigation. For example, the connections of association hasWork, corresponding to classes Artist and Work, are called creator and creation (lines 17–18).

Classes can have a primary key, composed of one or more attributes. This specifies that the given subset of the attributes uniquely identifies an instance of the class. In our example, as a gross simplification, title serves as a key in class Work, i.e. there cannot be two works (books, for example) with the same title.

Finally, invariants can be specified for classes and associations using the object constraint extension of UML, the OCL language [22]. Invariants give statements about instances of classes (and links of associations) that hold for each of them. The constraint in the declaration of Artist (line 5) is an invariant stating that the publication date of each work of an artist is greater than 1900$^2$. The identifier self refers to an arbitrary instance of the context, in this case the class Artist. Then two navigation steps follow. In the first step we navigate through the association hasWork to an arbitrary piece of work of the artist, while in the second step we go from the work to its publication date, and finally state that this date is always greater than 1900.

In addition to the object oriented modelling paradigm, the SILan language also supports constructs from the Description Logic world. This recently added feature of SINTAGMA is discussed in Section 3.3.

---

1. In SILan double colons (::) separate the model name from the name of its constituent (class, association, etc.).
2. This may be so because the underlying information sources are known to be dealing with works of art of 20th century or later.
Abstractions

For mediation, we need mappings between the different sources and the integrated model. These mappings are called abstractions because they often provide a more abstract view of the notions present in the lower level models. An example abstraction called $w_0$ can be seen in Figure 3.3.

```
abstraction w0 (m0: Interface::Product,
    m1: Interface::Description
    -> m2: Art::Work) {
  constraint
    m1.id = m0.id and
    m1.category = "artwork"
  implies
    m2.title = m0.name and
    m2.author = m0.creator and
    m2.date = m0.creation_date and
    m2.type = m1.subcategory and
    m2.DL_ID = m0.name;
}
```

Figure 3.3: SILan representation of the abstraction populating class Work

This abstraction populates the class Work (cf. Figure 3.2) in the model Art using classes Product and Description, both from the model Interface (lines 1–3). This means that the abstraction specifies how to create a “virtual” instance of class Work, given that the other two classes are already populated (e.g. they correspond to real information sources). In lines 1–3 the identifiers $m_0$, $m_1$ and $m_2$ are declared, and these will be used throughout the abstraction specification to denote instances of the appropriate classes.

The abstraction describes that given an instance of class Product called $m_0$ and an instance of class Description called $m_1$, for which the conditions in lines 6–7 hold, there exists an instance $m_2$ of class Work with attribute values specified by lines 9–13. Note that line 6 specifies that the id attributes of the two instances have to be the same, and thus corresponds to a relational join operation. In our integration scenario (see Section 3.5) Product and Description actually correspond to real-world Oracle tables containing various products and their descriptions, including books and paintings.

These two sources share the key id (line 6). While the first one supplies four fields to Work objects (title, author, date and DL_ID), the contribution of the second one is a single field (type). However, this second table has information to ensure that only relevant products (works of art) are included in class Work, through the condition in line 7.

We note that other abstractions can also populate class Work. In this case the set of instances of Work will be the union of the instances produced by the appropriate abstractions. Note that if a new information source is added, we only have to specify a new abstraction corresponding to this source, while the existing abstractions do not have to be modified.

Notice that the abstraction in Figure 3.3 takes the form of an implication describing how the given sources can contribute to populating the high level class Art::Work. This is characteristic of the Local as View integration approach [17].

3.2.2 The Wrappers

Wrappers provide a common interface for accessing various information source types, such as relational and object-oriented databases, semi-structured sources (e.g. XML or RDF), as well as Web-services.

---

3Attribute DL_ID comes from the class DLAny, of which class Work is a descendant. It has a special role, as explained in Section 3.3.3.
A wrapper has two main tasks. First, it extracts meta-data from the information source and delivers these to the Model Manager in the form of SILan models. For example, in case of relational sources, databases correspond to models, tables to classes, columns to attributes, as shown in Figure 3.4.

The other principal task of a wrapper is to transform queries, formulated in terms of this interface model, into the format required by the underlying information source, and thus allow for running queries on the sources.

3.2.3 The Mediator

The Mediator [5] supports queries on high level model elements by decomposing them into interface model specific questions. This is performed by creating a query plan satisfying the data flow requirements of the sources. During the execution of this query plan the data transformations described in the abstractions are carried out. Whenever we query a model element in SINTAGMA, the Model Manager provides the following two kinds of information to the Mediator:

1. the query goal itself, i.e. a Prolog term representing what to query;
2. set of mediator rules, using which the Mediator can decompose the complex query into primitive ones (i.e. queries that refer only to interface models).

For example, let us consider the query shown below involving class Work.

query RecentWork
select *
from w: Art::Work
where w.date > 2000;

This query is looking for recent works, namely those instances of the class Art::Work that were created after 2000\(^4\). In this case, the query goal is similar to the following simple Prolog expression:

\[
\text{:- } \text{Work:} \text{class:220'} (\text{DT}, \{A, B, C, D, E\}, \text{DA}), C > 2000. \quad (3.1)
\]

Here, the first Prolog goal retrieves an instance of Art::Work. The variables in this term will be instantiated during query execution. The predicate name ‘Work: class: 220‘ is a concatenation of three strings: the kind of the model element (class) and its unique internal identifier (220), preceded by the unqualified—and thus non-unique—SILan name (Work), provided for readability. Model elements are often referred to by handles of form \text{Kind(Id)}, e.g. \text{class (220)}. Note that the above predicate name represents the static type of the instances queried for, as opposed to the dynamic type which can be different, if the returned object belongs to a descendant class of Work.

\(^4\)We could have created a class named RecentWork and populated it by an appropriate abstraction. Then, instead of formulating a SILan query, we could have simply directly asked for the instances of this class. The question whether to use a query or an abstraction is a modelling decision.
The dynamic type of the queried instance, i.e. the handle of the most specific class it actually belongs to, is returned in the first argument of the goal. The second argument contains the values of the static attributes, in this case we have five such variables (cf. declaration of class Work in Figure 3.2). For example, \( C \) denotes the value of the attribute date. The third and last argument of the query term carries the values of the dynamic attributes. These represent the additional attributes (not known at query time) of the instance if it happens to belong to a descendant class of Art::Work.

The second part of the query goal corresponds to a simple arithmetic OCL constraint, which uses variable \( C \) representing the date attribute of the work in question.

The mediator rules representing the abstraction \( w_0 \) shown in Figure 3.3 take the following form:

\[
\begin{align*}
\text{Product:} & \text{class:}190\text{'(\_,[Title,Id,Author,Date],\_)}), \\
\text{Description:} & \text{class:}191\text{'(\_,["artwork",Id,Type],\_)} \rightarrow \\
\text{Work:} & \text{class:}220\text{'(class(220),[Title,Title,Author,Date,Type],[])}
\end{align*}
\]

The specific rule above describes how to create an instance of Work whenever we have two appropriate instances of classes Product and Description available. If there were more abstractions, the Mediator would get more rules as there would be more than one possible way to populate the given class.

Note that the mediator rules are also used to describe inheritance between model elements. In such a case the dynamic type of the model element on the right hand side of the rule is a variable (as opposed to the constant \( \text{class}(220) \) above). This variable is the same as the dynamic type of the model element on the left hand side. The dynamic attributes are propagated similarly.

Finally, let us state that an n-ary association is implemented as an n-ary relation, each argument of which is a ternary structure corresponding to a class instance, similar to the first goal of (3.1). For example, a query goal for the association hasWork (cf. Figure 3.2) has the following form:

\[
\text{:- } \text{hasWork:association:}227\text{'}( \\
\text{’Artist:} & \text{class:}218\text{'(\text{DT1},[DL_ID1,Name,Birthdate],\text{DA1})}, \\
\text{’Work:} & \text{class:}220\text{'(\text{DT2},[DL_ID2,Title,Author,Date,Type],\text{DA2})})
\]

\[
(3.2)
\]

\[The specific rule above describes how to create an instance of Work whenever we have two appropriate instances of classes Product and Description available. If there were more abstractions, the Mediator would get more rules as there would be more than one possible way to populate the given class.

Note that the mediator rules are also used to describe inheritance between model elements. In such a case the dynamic type of the model element on the right hand side of the rule is a variable (as opposed to the constant \( \text{class}(220) \) above). This variable is the same as the dynamic type of the model element on the left hand side. The dynamic attributes are propagated similarly.

Finally, let us state that an n-ary association is implemented as an n-ary relation, each argument of which is a ternary structure corresponding to a class instance, similar to the first goal of (3.1). For example, a query goal for the association hasWork (cf. Figure 3.2) has the following form:

\[
\text{:- } \text{hasWork:association:}227\text{'}( \\
\text{’Artist:} & \text{class:}218\text{'(\text{DT1},[DL_ID1,Name,Birthdate],\text{DA1})}, \\
\text{’Work:} & \text{class:}220\text{'(\text{DT2},[DL_ID2,Title,Author,Date,Type],\text{DA2})})
\]

\[
(3.2)
\]

### 3.3 Conceptual modelling in SINTAGMA

Let us now introduce the new DL modelling capabilities of the SINTAGMA system. First we discuss why we need Description Logic models during the integration process and provide an introductory example. Then we present the DL constructs supported by our system and discuss the restrictions we place on their usage. Finally, we summarise the tasks of the integration expert when using DL elements during integration.

#### 3.3.1 An introductory example

In the Model Warehouse we handle models of different kinds. We distinguish between application and conceptual models. The application models represent existing or virtual information sources and because of this they are fairly elaborate and precise. Conceptual models, however, represent mental models of user groups, therefore they are vaguer than the application models.

Our experience shows that to construct such models it is more appropriate to use some kind of ontological formalism instead of the relatively rigid object oriented paradigm. Accordingly, we have extended our modelling language to incorporate several description logic constructs, in addition to the UML-like ones described earlier. In the envisioned scenario, the high-level models of the users are formulated in description logic and via appropriate definitions they are connected to lower-level models. Mediation for a conceptual model follows the same idea we use for any other model: the query is decomposed, following the definitions and abstractions, until we reach the interface models (in general, through some further intermediate models) which can be queried directly.
Before going into the details, we show an example to illustrate the way how DL descriptions are represented in SILan (note that Writer and Painter are both descendants of class Artist, but otherwise they are normal UML classes; we will present more details about these classes in Section 3.5).

model Conceptual {
  class WriterAndPainter {};
  constraint equivalent {
    WriterAndPainter,
    Unified::Writer and Unified::Painter};
};

Here we define the class WriterAndPainter by providing a SILan constraint. This constraint can be placed anywhere in the Model Warehouse: in the example above we simply put it in the very model that declares the class WriterAndPainter itself. The constraint actually corresponds to a DL concept definition axiom: WriterAndPainter \equiv Writer \cap Painter. Namely, it states that the instances of class WriterAndPainter are those (and only those) who belong to the unnamed class containing the individuals who are both writers and painters. Thus, description logic concepts are defined using the Global as View approach [17], as opposed to the Local as View techniques applied in populating high-level classes using abstractions (cf. Section 3.2.1).

Note that the class WriterAndPainter could be created without DL support. However, in that case the integration expert would have to go through a much more elaborate process of creating the high level class WriterAndPainter, specifying all its attributes and populating it with an appropriate abstraction. This abstraction would have to implement the constraint (3.3), through an appropriate join-like operation.

Now, with DL support, the expert formulates a very short and intuitive DL axiom. We argue that this is easier for the expert to do, and it also makes the content of the Model Warehouse more readable to others.

### 3.3.2 Description logic elements in SILan

From the DL point of view, SINTAGMA supports acyclic Description Logic TBoxes containing only concept definition axioms, which are formulated in an extension of the \( \mathcal{ALCN} (D) \) language (see more below about the extension). Only single atomic concepts, so called named symbols can appear on the left hand side of the axioms, such as WriterAndPainter in example (3.3). The remaining atomic concepts, not appearing on the left hand side are called base symbols. Such a TBox is definitorial, i.e. the meaning of the base symbols unambiguously defines the meaning of the named symbols. The base symbols, in our case, correspond to normal SINTAGMA classes and associations, e.g. Writer and Painter in the example (3.3). The ABox is a set of concept and role assertions, as determined by the instances of the classes which correspond to the base symbols participating in the TBox.

The DL concept constructors supported by SINTAGMA and their SILan equivalents are summarised in Table 3.1. Note that this table actually describes the possible concept formats on the right hand side of a definition axiom, assuming that we have expanded the TBox\(^5\).

The only non-classical DL element in Table 3.1 is the concrete domain restriction (the last line in the table). Such a restriction specifies a subset of instances of the base concept A for which the given OCL constraint holds. This is a generalisation of the idea of concrete domains in the Description Logics world. Below we show an example of a concrete SILan restriction describing those works whose type (i.e. the value of the attribute type) is “painting”.

class constraint Art::Work satisfies self.type="painting"

The reason we allow only concept definition axioms is that we aim to use DL concepts to describe executable high-level views of information sources. In this sense a DL concept is actually a syntactic variant of a SILan query or a SILan class populated by an abstraction.

---

\(^5\)The expanded version of an acyclic TBox is obtained by repeatedly replacing every named symbol on the right hand side of an axiom by its definition. This process is repeated until no further named symbols are left on the right hand side. The fact that the TBox is acyclic ensures the termination of this process.
Note that this also implies that we use the Closed World Assumption (CWA) in DL query execution. We argue that this is appropriate because of the following three reasons. First, CWA automatically ensures that our DL constructs are semantically compatible with other constructs in the SINTAGMA system. Second, we argue that the Open World Assumption (OWA) is applicable when we have only partial knowledge and would like to determine the consequences of this knowledge, true in every universe in which the axioms of this partial knowledge hold. In contrast with this, in the context of information integration, our users would like to consider a single universe, in which a base concept or a role denotes exactly those individuals (or pairs of individuals) which are present in the corresponding database. To illustrate this issue, let us consider the following example: the concept of novice painter is defined to contain painters having at most 5 paintings (for example, being a novice painter may be a precondition for a government grant). To model this situation, the integration expert creates the DL axiom shown below.

\[
\text{NovicePainter} \equiv \text{Painter} \sqcap (\leq 5 \text{hasPainting})
\]

However, querying this concept, using OWA, will provide no results in general, as an open world reasoner would return an individual only if it is provable that it has no more than 5 paintings. Practically, this is not what the information expert wants.

The third reason why we decided to use the closed world assumption is that by using CWA we can implement DL queries using the well researched, efficient database technology.

### 3.3.3 Modelling methodology and tasks of the integration expert

The integration expert is responsible for creating the DL axioms. Although these are represented in SILan within the SINTAGMA system, the expert can use any available OWL editor to create OWL descriptions. These descriptions then can be loaded by the OWL importer of the SINTAGMA system that basically realises an OWL-SILan translation (cf. the “Model Im(Ex)port” box in Figure 3.1).

One thing the expert should take care of is to match the names of the base symbols and the corresponding SINTAGMA classes and associations. This is often done in two steps: first the integration expert

<table>
<thead>
<tr>
<th>Name</th>
<th>DL Syntax</th>
<th>SILan equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base concept</td>
<td>$A$</td>
<td>UML class</td>
</tr>
<tr>
<td>Atomic role</td>
<td>$R$</td>
<td>UML association</td>
</tr>
<tr>
<td>Intersection</td>
<td>$C \sqcap D$</td>
<td>$C$ and $D$</td>
</tr>
<tr>
<td>Union</td>
<td>$C \sqcup D$</td>
<td>$C$ or $D$</td>
</tr>
<tr>
<td>Negation</td>
<td>$\neg C$</td>
<td>not $C$</td>
</tr>
<tr>
<td>Value restriction</td>
<td>$\forall R.C$</td>
<td>slot constraint $R$ all values $C$</td>
</tr>
<tr>
<td>Existential restriction</td>
<td>$\exists R.C$</td>
<td>slot constraint $R$ some value $C$</td>
</tr>
<tr>
<td>Number restriction</td>
<td>$\Join nR$</td>
<td>slot constraint $R$ cardinality $i..j$</td>
</tr>
<tr>
<td>Top</td>
<td>$\top$</td>
<td>DLAny</td>
</tr>
<tr>
<td>Bottom</td>
<td>$\bot$</td>
<td>DLEmpty</td>
</tr>
<tr>
<td>Concrete restriction</td>
<td>—</td>
<td>class constraint $A$ satisfies OCL</td>
</tr>
</tbody>
</table>

Table 3.1: DL-related constructs supported in SILan
creates concept definition axioms using the widely accepted terminology of the domain, not paying attention to the names of the model elements in the Model Warehouse. Next, the expert provides additional definition axioms for each base symbol connecting it with the proper model element. For example, we could use names A and B instead of Writer and Painter in (3.3), provided that we also encode in SILan the equivalents of the following DL axioms:

\[
A \equiv \text{Writer} \\
B \equiv \text{Painter}
\]

A further crucial issue is to decide how to identify the instances of the base concepts, e.g. the instances of the class Writer and class Painter. Without this, it is not possible to determine the instances of class WriterAndPainter.

In a traditional DL ABox, an instance has a name that unambiguously identifies it. In SINTAGMA, similarly to databases, an instance is identified by the subset of its attribute values. For example, two writers could be considered to be the same if their names match, assuming that name is a key in class Writer.

The problem is that such keys are fairly useless when we compare instances of different data sources. This is because, in general, we cannot draw any direct conclusion from the relation of the keys belonging to instances from different classes. For example, databases containing employees often use numeric IDs as keys. Having two employees from different companies with the same ID does not mean that we are talking about the same person. Similarly, if the IDs of the employees do not match, they are not necessarily different persons.

What we need is some kind of shared key that uniquely identifies the instances of the classes participating in DL concept definitions. Luckily, the object-oriented paradigm we use in SINTAGMA provides a nice way to have such identifiers.

We have mentioned that in SINTAGMA the notion of DL concept is a syntactic variant of SINTAGMA class. This also means that the result of a DL query is an ordinary instance that has to belong to some class(es). For example, when we are looking for the instances that are elements of both classes Writer and Painter we are actually interested in an artist instance belonging to these classes simultaneously. This is true in general: whatever DL concept constructs we use to describe a DL concept the result must belong to some class that is a common ancestor (in terms of inheritance) of the classes involved.

Instead of asking the integration expert to define such common ancestor classes in an ad hoc way, we introduce the built-in class DLAny. This class corresponds to the DL concept top (\(\top\)) and it has only one attribute called DL_ID, which is a key. We require that all the classes participating in DL concept definitions are the descendants of DLAny\(^6\) (cf. lines 2 and 8 of Figure 3.2). Because of the properties of inheritance, attribute DL_ID will be a key in all of the descendant classes, i.e. it will exactly serve as the global identifier we were looking for.

Now, the task of the integration expert is to assign appropriate values to the DL_ID attributes: she needs to extend the existing abstractions populating the base symbols (classes) to also consider the attribute DL_ID. By appropriate values we mean that the DL_IDs of two instances should match if these instances are the same, and should differ otherwise. An example for this can be seen in Figure 3.5 populating the class Writer, which is part of a bigger integration scenario to be shown later in Section 3.5.

This abstraction populates the class Writer from an interface class called Member (lines 1–2), which represents a membership database of an imaginary “International Writer Association” (IWA). Let us assume that the members of this association have some kind of a unique identifier, such as the membership number, present in the underlying database. It may be worth bringing this key to the class Writer (line 7) as it makes possible to find writers efficiently if they happen to be IWA members. However, the unique identifier from the DL point of view has to be different: in fact it is the concatenation of the first and last name of the writer, with a space in between (lines 4 and 9).

This is because the class Writer can also be populated from other sources (e.g. Person, see Figure 3.8) where the IWA number makes no sense and so the member_id attribute is set to "n/a". Furthermore, we may want class Writer to be a descendant of class Artist (cf. Figure 3.8), together with some other

---

\(^{6}\text{Note that this is a necessary condition. As for any concept } C, C \sqsubseteq \top \text{ holds, any DL instance has to belong to the class corresponding to } \top, \text{ i.e. to} \text{DLAny.}\)
classes, such as Painter. This requires a key that can be computed from all the underlying sources, such as the name of the artist\(^7\).

To summarise, the integration expert has to perform the following tasks when DL modelling is used during the integration process:

1. declare DL classes and for each provide corresponding definition axioms;
2. ensure that each base concept appearing in the definition axioms is:
   - (a) inherited from class DLAny,
   - (b) populated properly, i.e. its DL_ID attribute is filled appropriately.

### 3.4 Querying conceptual models in SINTAGMA

Now we turn our attention to querying DL concepts in SINTAGMA. As described in Section 3.2.3 our task is to create a query goal and a set of mediator rules. When we query a DL class, mediator rules are only generated for the base symbols. As these are ordinary classes and associations, this process is exactly the same as the one we use for cases without any DL construct involved. This means that we can now focus on the construction of the query goal.

Recall that a SINTAGMA instance is characterised by three properties, as exemplified by (3.1) on page 63: its dynamic type DT, its static attributes SA and its dynamic attributes DAs. Below we will use the variable name As to denote the full attribute list of an instance, i.e. the concatenation of the static and dynamic attribute values, with the exclusion of DL_ID.

A DL class has only a single static attribute, the DL_ID key. However, in contrast with an object oriented query, a DL query may return an answer that has multiple dynamic types. For example, when we enumerate the class WriterAndPainter we get instances that belong to both classes Writer and Painter (something which is not possible in the standard UML modelling). Accordingly, an answer to a DL query takes the form of a pair \((\text{ID}, \text{DTA})\), where ID is the DL_ID\(^8\) containing the unique name of the DL instances (see Section 3.3.3), while DTA is a Prolog structure containing the dynamic types of the answer, each paired with the corresponding full attribute list. The DTA structure is thus either a single DT-As pair, or recursively, two DTA structures joined using the comma operator: \((\text{DTA}_1, \text{DTA}_2)\).

Figure 3.6 describes the mapping from an arbitrary DL concept expression to the corresponding query goal. Here we define a function \(\Phi_C\) which, given an arbitrary concept expression \(C\), returns the corresponding query goal with two arguments, ID and DTA. We define this function by considering the DL concept constructors, as listed in Table 3.1.

Let us consider the cases one by one. If we have a base class, we simply create a query term representing the instances of the class, similar to the one in goal (3.1) and then convert the attributes retrieved to the

---

\(^7\)This is also a simplification. More realistically, the key could be the name together with the birth date.

\(^8\)We use the name ID instead of DL_ID for conciseness.
required form (DTA). Here operation ⊕ denotes the compile time concatenation of lists\(^9\), while \(A^N\) stands for the predicate name corresponding to concept \(A\). For example, \(\text{Work}^N = \text{Work:class:220}\), cf. (3.1) on page 63. Note that in the second argument of the query goal \(A^N\) we make use of the fact that the DL_ID attributes are always placed first in the static attribute list of an instance.

If we have the intersection of two concepts \(C\) and \(D\), we recursively transform concepts \(C\) and \(D\) and put them in a Prolog conjunction. The DTA structure is built from the structures recursively obtained from the execution of the transformations of concepts \(C\) and \(D\). Note that the resulting structure may contain duplicates, i.e. the same DT-As pair may be found in DTA more than once. These duplicates are only removed at the top level, i.e. when the final result of a query is presented. The transformation of union concepts is similar to the intersection: we create a Prolog disjunction.

Negation \(\neg C\) is implemented by using the Prolog negation-as-failure. This translation is only capable of checking whether a given instance with ID belongs to concept \(C\) or not. As usual in the database context, we restrict the use of negation to cases where negated queries appear only in conjunction with at least one non-negated query. In terms of DL concept expressions this means that negated concepts have to appear either in the scope of a quantifier, or in an intersection together with at least one non-negated concept. It is the task of the Mediator to find an appropriate order in the query plan where negation appears in a place where ID is instantiated [16]. The Mediator refuses to execute the query if such an order does not exist.

The next two cases involve associations. On the right hand side of these formulae \(R^N\) denotes the predicate name corresponding to the association itself. \(R_D\) (\(R_R\)) denotes the base class that is the domain (range) of association \(R\). Correspondingly, \(R^N_D\) and \(R^N_R\) stand for the predicate names of the classes \(R_D\) and \(R_R\), respectively\(^{10}\). Recall that a binary association is represented by a binary relation with ternary structures as arguments, as in (3.2).

The existential restriction \(\exists R.C\) is transformed to a query of the association \(R\) and the concept \(C\).

The goal corresponding to a value restriction \(\forall R.C\) first enumerates the domain of \(R\) and then uses double negation to ensure that the given instance has no \(R\)-values which do not belong to \(C\). Note that

\[^9\text{The } \oplus \text{ operator is used only with a static attribute list (SAs). For any base class, the length of the corresponding SAs is fixed (the number of static attributes excluding the DL_ID). Therefore, the SAs } \oplus \text{ DAs concatenation can be carried out at compile time.}\]

\[^{10}\text{For example, if } R = \text{hasWork}, \text{ cf. Figure 3.2, then } R^N = \text{'hasWork:association:227'}, R^N_D = \text{'Artist:class:218'} \text{ and } R^N_R = \text{'Work:class:220'}.\]
Φ_{\neg C}(ID2, \_\_) is invoked only when ID2 is already instantiated.

A number restriction \( [\exists nR] \) is transformed into a goal which uses the Prolog built-in predicate bagof [23] to enumerate the instances in the domain of \( R \) together with the number of \( R \)-values connected to them, and then simply applies the appropriate arithmetic comparison.

The last two lines of Figure 3.6 define the transformation of the top and bottom concepts. \( \top \) is mapped into \( \text{true} \), while \( \bot \) to \( \text{false} \). Querying these concepts on their own does not make sense, but these mappings are useful when transforming DL concepts such as \( \exists R \top \) or \( \forall R \bot \).

Having described the transformation of DL concepts to query goals, we now deal with the only remaining construct: the concrete restriction. A concrete restriction involving a base concept \( A \) and an OCL constraint \( O \) is transformed in a straightforward way into the query goal as shown below\(^{11} \):

\[
\Phi_A(ID, DTA), \ DTA = DT-AT, \ \Psi_O(ID, AT)
\]

To illustrate the general algorithm, two example transformations are presented in Figure 3.7. The first one shows the translation of the WriterAndPainter class described in (3.3) on page 65. The query goal is a conjunction that consists of three goals. The first two goals enumerate the instances of classes Writer and Painter with a condition that their \( ID \) attributes match. At this point we have identified those instances who are writers and painters at the same time. The last goal constructs the structure \( DTA \), describing the dynamic types and the corresponding attribute values of the given instances.

In the second example we look for a writer who has at least one piece of modern work. This DL concept involves the association \( \text{hasWork} \) and a class \( \text{Modern} \) (representing, say, contemporary pieces of art). The query goal becomes a bit more complex than in the first example: now it consists of four goals. The first goal enumerates the instances of class Writer. The second and the third goals filter out those writers that do not have any modern works. Here we have used the facts that the domain of \( \text{hasWork} \) is the class \( \text{Artist} \) and the range is the class \( \text{Work} \) (cf. Figure 3.2). Finally, the last goal builds the structure \( DTA \).

\[
\begin{array}{l}
\text{Class to query: WriterAndPainter} \\
\text{DL definition:} \quad \text{Writer } \sqcap \text{ Painter} \\
\text{Query goal:} \quad 'Writer: class: 234'(DT1, [ID, Name, Birth, IWA, Style], DA1), \\
\quad 'Painter: class: 236'(DT2, [ID, Name, Birth, Colour], DA2), \\
\quad DTA = (DT1-[Name, Birth, IWA, Style| DA1], \\
\quad \quad DT2-[Name, Birth, Colour| DA2])
\end{array}
\]

\[
\begin{array}{l}
\text{Class to query: ModernWriter} \\
\text{DL definition:} \quad \text{Writer } \sqcap \text{ \hasWork.Modern} \\
\text{Query goal:} \quad 'Writer: class: 234'(DT1, [ID, Name, Birth, IWA, Style], DA1), \\
\quad '\hasWork: association: 227' ( \\
\quad \quad 'Artist: class: 218'(DT2, [ID, Name, Birth], DA2), \\
\quad \quad 'Work: class: 220(_, [ID2|\_], \_), \\
\quad \quad 'Modern: class: 237'(_, [ID2|\_], \_), \\
\quad DTA = (DT1-[Name, Birth, IWA, Style| DA1], \\
\quad \quad DT2-[Name, Birth, Colour| DA2])
\end{array}
\]

Note that if a writer has more than one piece of modern work, the transformation in Figure 3.7 enumerates the writer multiple times. This is because the second goal can succeed more than once, leaving a choice point [115]. In the present version of SINTAGMA these duplicates are removed at the top level only, before the query results are presented to the user. In future, we will consider a more efficient solution, utilising the Prolog pruning operators (conditionals or cuts) to eliminate the unnecessary choices.

\(^{11}\Psi_O(ID, AT)\) denotes the Prolog translation of the OCL constraint \( O \). This is a feature which has already been present in earlier versions of SINTAGMA, before the introduction of the DL extensions, see [11].
Also note that in our example scenario attributes Name1, Name2 and Birth1, Birth2 will be instantiated to the same values, i.e. to the name and birth date of the modern writer. This is the consequence of the data representation we use in SINTAGMA, i.e. if an instance has multiple dynamic types, for each of them we supply all the attribute values.

3.5 A case study: artists

In this section we present a simple use case, where we focus on illustrating the DL extension of SINTAGMA. More complex traditional integration problems solved using SINTAGMA are discussed in other papers, for example in [86]. Figure 3.8 shows the content of our example Model Warehouse. Here we have four models on different abstraction levels.

![Figure 3.8: Content of the Model Warehouse](image)

The lowest one, Interface, contains classes directly corresponding to the information sources we aim to integrate. Class Member corresponds to some database table containing information about writers (members of a certain writers association), Person is the model of an XML source describing people (some...
of whom are possibly writers). We also have here class Exhibitor containing people some of whom are painters, and class Product containing art works among other products together with class Description which provides some information on products. These models are constructed automatically by different wrappers of the SINTAGMA system.

The next, more abstract model, called Unified, contains two classes Writer and Painter, their SILan descriptions are shown in Figure 3.9 (referring to class Artist introduced in Figure 3.2 in page 61). These classes provide a unified view of writers and painters over our heterogeneous information sources, i.e. querying Writer and Painter gives us all the known writers and painters respectively. These classes are populated by SILan abstractions: Writer by two, while Painter by only one. We can later extend our Model Warehouse to include more information sources on painters. This way Painter would also be populated by several abstractions. Please note how flexible this approach is: whenever we would like to add a new information source, all we have to do is to provide a new abstraction. This is fundamentally different from the way views are created in traditional database systems.

```plaintext
model Unified {
    class Writer: Art::Artist {
        attribute Integer member_id;
        attribute String style;
    };
    class Painter: Art::Artist {
        attribute String favourite_colour;
    };
}
```

Figure 3.9: SILan description of classes Writer and Painter

The third model Art describes an even higher view of the underlying information sources. It contains two classes connected by an association. Class Artist is declared to be the generalisation of classes Writer and Painter, i.e. Artist is a common “parent” of Writer and Painter, in terms of inheritance. Accordingly, it contains the union of the instances of these classes. Class Work incorporates works (books and paintings). In the example, Work is populated by only one abstraction. Association hasWork connects instances in class Artist with those in class Work, i.e. it allows us to navigate from an artist to her works. This association is populated by an abstraction (not shown in Figure 3.8) by creating virtual pairs from those instances of classes Artist and Work where the author of the work matches name of the artist.

Note that there is one more association in the Model Warehouse, called hasPainting. This association connects painters with their paintings and goes between different models. Similarly to hasWork this association is also populated by an abstraction, not shown here. Association hasPainting is used in the definition of PainterWriter (see below).

Up until now we have used the traditional features of SINTAGMA: classes, associations, generalisations, abstractions. Now we turn to the most abstract model, named Conceptual, which provides an even higher-level view of the information than the previous model.

The model Conceptual represents the knowledge of our specific example domain, in the form of DL concept definition axioms. These axioms form a simple ontology, a part of which is shown in Figure 3.10. This ontology talks about special types of artists, painters and writers. It states that a novice painter is a painter who has only painted no more than 5 paintings (axiom 1). Somebody is mostly writer if she is an artist who has produced at least 3 works, but has at most one painting (axiom 2). A productive writer has created at least 10 works (axiom 3). Somebody is painter-writer is she is a writer who has some paintings (axiom 4). Finally, a novelist is somebody who is only writing novels (axiom 5).

In practice, such an ontology can be created by the information expert manually or can be imported from an existing ontology using the OWL importer component of the SINTAGMA system. In SINTAGMA this
ontology is represented by a model containing classes with no attributes, together with the corresponding SILan constraints as shown below:

```plaintext
model Conceptual {
    class NovicePainter {}; class MostlyWriter {}; class ProductiveWriter {}; class PainterWriter {}; class Novelist {}; ...

    constraint equivalent {
        NovicePainter, Painter and {slot constraint hasPainting cardinality 0..5}
    }
    ...
};
```

Let us consider the base concepts used in our concept definitions in Figure 3.10. Most of these (i.e., Painter, Writer and Artist), appear in the underlying UML models. However, there is the concept of Novel, which has no direct UML counterpart. This concept can be defined using a concrete restriction of SILan, as shown below.

```plaintext
constraint equivalent {
    Novel,
    {class constraint Art::Work satisfies self.type="novel"}
};
```

This concludes the description of our example models. Having encoded our DL axioms in terms of SILan constraints, we can now execute DL queries. For example, we can ask SINTAGMA to enumerate the instances of class ProductiveWriter. This query will produce instances similar to the following:

```plaintext
({'Lisa James',
   ['Writer']->['Lisa James', 1965, 42, 'fantasy'],
   ['Painter']->['Lisa James', 1965, 'red']})
```

Here, the string ‘Lisa James’, appearing in line 1, corresponds to the ID of Figure 3.6, i.e. the shared DL identifier. Lines 3–4 contain the list of the dynamic types and corresponding attributes of the instance. This specific instance has two dynamic types: she is a writer and a painter at the same time (lines 3 and 4). As a writer, she has a name, birth date, her membership ID and a style attribute. As a painter we also know her favourite colour.
3.6 Related work

The two main approaches in information integration are the Local as View (LAV) and the Global as View (GAV) [17]. In the former, sources are defined in terms of the global schema, while in the latter, the global schema is defined in terms of the sources (similarly to the classical views in database systems). Information Manifold [72] is a good example for a LAV system. Examples for the GAV approach include the Stanford-IBM integration system TSIMMIS [20], and the DL based integration system Observer [98].

In SINTAGMA we apply a hybrid approach, i.e. we use both LAV and GAV. When using abstractions to populate high-level classes we employ the LAV principle, while in case of Description Logic class definitions we use the GAV approach.

There are several completed and ongoing research projects in the area of using description logic-based approaches for both Enterprise Application Integration (EAI) and Enterprise Information Integration (EII). The generic EAI research stresses the importance of the Service Oriented Architecture, and the provision of new capabilities within the framework of Semantic Web Services. Examples for such research projects include DIP [55] and INFRAWEBS [45]. These projects aim at the semantic integration of Web Services, in most cases using Description Logic based ontologies and Semantic Web technologies. Here, however, Description Logic is used mostly for service discovery and design-time workflow validation, but not during query execution.

On the other hand, several logic-based EII tools use Description Logic and take a similar approach as we did in SINTAGMA. That is, they create a DL model as a view over the information sources to be integrated. The basic framework of this solution is described e.g. in [18, 15]. The fundamental difference with our approach is that these applications deal with the classical Open World Assumption, as already discussed in Section 3.3.2.

On the theoretical side an interesting description logic is the $\mathcal{ALC}_K$ [26] which adds a non-monotonic $K$ operator to the $\mathcal{ALC}$ language to provide the ability to use both the CWA and the OWA, when needed. $\mathcal{ALC}_K$ has several implementation, the Pellet reasoner [122], for example, supports this logic. However, $\mathcal{ALC}_K$ lacks the ability to express cardinality constraints, which is a feature frequently used in information integration scenarios.

Finally, we mention that the Description Logic Programming (DLP) approach, first introduced in [46], also employs the idea of translating DL axioms into Prolog goals (cf. the approach summarised in Table 3.6). In contrast with our approach DLP uses the Open World Assumption and does not deal with negation and cardinality restrictions.

3.7 Conclusion

In this chapter we have presented the DL extension of the information integration system SINTAGMA. This extension allows the information expert to use Description Logic based ontologies in the development of high abstraction level conceptual models. Querying these models is performed using the Closed World Assumption over the underlying information sources.

We have presented the main components of the SINTAGMA system: the Model Manager which is responsible for maintaining the Model Warehouse repository, the Wrapper, which provides a uniform view over the heterogeneous information sources and the Mediator, which decomposes complex high-level queries into primitive ones answerable by the individual information sources.

Next, we have described the newly introduced DL modelling elements the integration expert can use when building conceptual models and we have also discussed the modelling methodology she has to follow. We have defined a transformation of DL queries to Prolog goals, used in the SINTAGMA system for DL query execution. We have also illustrated our approach by providing a use case about artists.

We believe that because Description Logics are not expressive enough to be used alone for solving complex modelling problems, some kind of hybrid techniques are necessary. We argue that our solution for combining DL and UML modelling in a unified integration framework provides a viable alternative to existing systems. The usage of DL constructs in building high-level conceptual models has substantial benefits, both in terms of modelling efficiency and maintenance.
Part II

Case studies in semantic applications
Chapter 4

Plagiarism detection in programs

The chapter presents a plagiarism detection framework the goal of which is to determine whether two programs are similar to each other, and if so, to what extent.

The issue of plagiarism detection has been considered earlier for written material, such as student essays. For these, text-based algorithms have been published. We argue that in case of program code comparison, structure based techniques may be much more suitable. The main idea is to transform the source code into mathematical objects, use appropriate reduction and comparison methods on these, and interpret the results appropriately.

We have designed a generic program structure comparison framework and implemented it for the Prolog and SML programming languages. We have been using the implementation at BUTE to successfully detect plagiarism in homework assignments for years.

4.1 Introduction and motivation

Comparison of essays and other written materials has been in focus in recent years [127]. Detecting plagiarism in written materials is an issue in education as well as in law procedures. World wide public polls show that two-thirds of university students have used other people’s ideas in an impermissible way at least once during their studies. Law disputes include the SCO-IBM debate over the allegedly unauthorised use of portions of the AIX operating system in Linux.

Regrettably, several sites on the Internet provide free or low cost, quick and efficient access to written materials of many types. Unbelievably, sites such as CheatHouse\(^1\) or SchoolSucks\(^2\) proudly provide tons of essays, dissertations, reports, etc. for students looking for an easy way to have their assignment of some sort fulfilled. We do agree that it is a good idea to get acquainted with the area one is interested in by reading similar materials. However inspiring someone to cheat is a different issue.

In case of programming assignments, it is important to detect the duplication of programs or parts of these. Students attending the course “Declarative Programming” at BUTE are expected to hand in a major programming assignment during the semester. This means mass amount of program sources year by year.

Checking these programs by hand seems to be beyond possibility. Having \(n\) programs we should check \(\frac{n(n-1)}{2}\) pairs to have all the cases covered. Notice, that we really should check all of the pairs, because the relation “P1 is similar to P2”, where P1 and P2 are programs, is not transitive. This practically means that even if we know that source A is similar to source B and source B to source C we cannot draw any direct conclusion about the similarity degree of sources A and C.

Luckily, in our particular case several assignments can be excluded from the whole set. For example, we do not care whether two bad solutions are similar or not (a solution is bad if it does not solve a certain percentage of the given test cases). However we still have \(O(n^2)\) pairs to test manually, where \(n\) is often greater than 100.

\(^1\)http://www.cheathouse.com
\(^2\)http://www.schoolsucks.com
Our aim was to develop methods and tools to assess the similarity of programs in order to narrow down the need for manual testing to an acceptable amount. We have defined the notion of a similarity degree which reflects how much two programs match. For the methods to be generic and flexible enough we have developed a multi phase comparison framework.

The actual comparison is performed between mathematical entities where the meaning of similarity can be formally specified. These entities are generated from the programs to be compared. The procedure may vary for different programming languages, so separate front-end modules should be developed for each language. Naturally, the mathematical entities must be generic and powerful enough to be applicable to different languages. We have chosen directed, labelled graphs for this purpose. Now, the comparison of source programs is actually reduced to calculating the similarity measure of graphs. Notice, that this way it is also possible to determine the similarity degree of two programs written in different languages.

The framework is customisable, so that it remains usable under varying circumstances. For instance, in case of shorter programs a different similarity threshold may be more appropriate than in the case of bigger ones. Moreover, we found that applying certain well selected simplifying graph transformations, called reductions, has favourable effects on the efficiency of the approach. Such reductions include removing specific nodes and edges and thus creating higher level, more abstract views of the programs.

The structure of the chapter is as follows. In Section 4.2 we give a brief comparison of our approach with other ongoing research work. Next, we describe what we expect from a plagiarism detection framework, i.e. what are the types of student tricks it should be resistant to. In Section 4.4 we give an overview of the proposed framework and introduce the main concepts. Following this, we describe the three components of the framework: the Front-end module, the Simplifier and the Comparator. Section 4.5 describes the prototype implementation of the framework for Prolog and SML programs. Next, we evaluate the system and show execution results. Finally, we give a summary of our work.

4.2 Related work

Several solutions exist for detecting plagiarism in written documents (like iThenticate [69], FindSame [68], CopyCatch [63], SCAM [120] or the new Hungarian portal from the Computer and Automation Institute of the Hungarian Academy of Sciences called KOPI [64]). However, this is not the case for program sources. A reason for this may be that it is widely believed that detecting plagiarism in programs is much easier than in free text. This is because programming languages are formally defined and, as opposed to the case of free text, it is generally assumed that people use only a few tricks to hide the fact of plagiarism.

Alan Parker and James Hamblen in [113] explicitly say that copied software is “a program which has been produced from another program with a small number of routine transformations”. These routine transformations include modifying the comments, changing the names of the variables or (in the worst case) changing the control structures (e.g. using while instead of for). The suggested technique for comparing programs is the following:

1. Get rid of every comment in the source codes.
2. Get rid of every useless new line, white space, etc.
3. For each pair of source programs use a normal UNIX diff program and compare the files line by line.
4. Examine the results.

In [34] J. A. Faidhi and S. K. Robinson suggested a scale which defines the level of plagiarism (L0-L6) based on what kind of modifications the cheater used. For example, we obtain L1 from L0 by modifying the comments, L2 from L1 by further modifying the variable names as well, etc. This scale is often used by programs for plagiarism detection to “position” themselves.

Most existing software solutions are based on statistical or lexicographic approach where, for example, they compare identifiers with identifiers to determine how similar the source programs are. Such systems are the DUP [6], SIM [47], SIFF [7] or Bandit [139].

On the other hand, approaches based on structural properties were already proposed several decades ago. For example, in [13] J. M. Bieman and N. C. Debnath suggested building program graphs, while
T. J. McCabe proposed [97] to compute a characteristic numeric value, a metric, for each program code according to its complexity (which was based on the number of computation paths available within the program). This metric is widely known today as cyclomatic complexity.

Further programs that support structure comparison include the Plague [140], the YAP (Yet Another Plague) series [141], and the Moss (Measure Of Software Similarity) [118] program. Plague builds so called structure profiles for source codes and compares them. The YAP programs implement a two phase approach. First they convert the source programs into a more unified form, e.g. removing comments, translating upper-case letters to lower case. In the second phase (depending on the actual YAP version) they apply algorithms, such as Heckel’s isolation technique [54], that are resistant to specific structural changes, e.g. changing the order of independent statements or replacing a procedure call by the procedure body. The authors of Moss have developed a general algorithm for calculating a so called fingerprint from an arbitrary document which they claim to be especially precise in case of source programs.

Paper [109] introduces an XML-based model called XPDec (XML Plagiarism Detection Model) suitable for programs written in a procedural language such as C or Pascal. XPDec uses XML to represent structural properties of the source programs and is useful for detecting common forms of reordering plagiarism. An extended version of this approach is presented in [110] which takes also into account the structure of the control sequences in the source programs.

Our approach introduced in this chapter belongs to the group of program plagiarism systems utilising the structural properties of the source programs. However, instead of providing sophisticated comparison techniques that are resistant to the most common tricks we apply so called reduction steps to create more abstract views of the programs. These views are then compared by using relatively simple comparison algorithms. We argue that this approach makes our system fairly efficient and easy to customise.

### 4.3 Goals

We now discuss the most significant student tricks we believe a plagiarism detection framework should be resistant to. To illustrate such tricks in a language independent way we use pseudo-language examples below. We realise that there exist tricks only applicable for specific programming languages. Handling these is the task of the concrete implementation of such a framework (cf. Section 4.4).

Changing the names of identifiers and variables is the most common trick. A piece of source code which contains only single letter variable names may look rather confusing and tangled. However, it can be easily transformed into a program which uses talkative names. For humans, sometimes only this is enough to hide the fact of plagiarism. A similar trick is to change the natural language in which the program identifiers are formulated: use English names in one program and use another language in the other. It is also possible to change not just the variable, but the function and/or predicate names, too. For example, it is very easy to transform the function

```c
void solve_the_problem(Input_data, Results) {...}
```

...to the following:

```c
void do(Input, Output) {...}
```

One can also change the number of arguments (the so called arity) of the functions, without affecting the code. For example one can use dummy parameters, which are set to something irrelevant at call time. If one changes not only the name of a function, but also its arity, it may become really difficult for the human to recognise that it is semantically equivalent to some other function.

Sometimes it is profitable for students to cut the code into several pieces and place them into separate files using the module system of the given language. Similarly, reordering the sequence of the function definitions in a source file is an easy, but often effective trick. Students also like to change the order of statements in the body of a function if these statements do not depend on each other: for example, two independent variable assignments can be switched. In case of logic programming languages, this kind of trick is very common as a body of a predicate is the logical conjunction of so called goals. This means that these goals can often be reordered freely without effecting the execution of the program.
Putting useless functions into the code may also be used to disguise plagiarism. For example we can “borrow” some code from another program which has nothing to do with the current programming assignment. Computer based methods may find this disturbing, because this technique introduces new variables and functions, changes the size of the file, etc. Sometimes one can recognise this trick by doing static source code analysis and detecting that these functions are never called, but this is not true in general.

Consider the following example, where the procedure calculate will never be called. This procedure can be anything, most likely a piece of some big code, with the only aim to conceal the fact that the original source code for solve_the_problem was made by some other individual.

```c
int solve_the_problem(A, B) {
    if (A > 0) {
        ...
        X = A + 35;
        ...
        if (X < 0) // X cannot be negative here
            calculate(X, B);
        else
            X = 2;
        ...
    }
    ...
}
```

In the general case those parts of the program which are never called can only be detected at runtime. Unfortunately, even if we detect such code fragments it does not mean that we found an instance of plagiarism. Sometimes such code is simply the result of programming errors, which even the author of the program is not aware of.

Analogously to placing useless procedures in the program code one can place useless calls in the body of a procedure without changing its task. In the following example we show two totally useless lines inserted into a function, not changing the execution of the program:

```c
... C = 2; A = 3-C; // A = 1 ...
... if (C == A+1) { // check if C = 2 ...
```

Finally, we show two tricky, but easily implementable types of program transformation. The first we named call-tunneling, while the second call-grouping. Call tunneling is based on the idea that instead of letting function A to call function C directly, we insert an intermediate function B. In this new scenario A calls B and B calls C. If function B returns what it got from C without any modification, then the transformed program will be equivalent to the original one. Call-tunneling is very hard to detect, because, for example, function B is actually called during the execution, therefore it seems to be an important part of the program.

Call-grouping is a simple technique to significantly modify the structure of a program even if one does not really understand what the code actually does. The main idea is very similar to that of call-tunneling: if there is a function which calls several others, we can regroup these calls into some new functions to produce a totally different code structure. Let us consider the following piece of code:

```c
int original_function(A, B) {
    T = call1(A);
    Q = call2(B, T);
    E = call3(Q);
    Z = call4(A, E);
    return call5(Z);
}
```
Using call-grouping one can transform it to the following equivalent program.

```c
int grouped_function(A, B) {
    E = temp1(A, B);
    return temp2(A, E);
}
```

```c
int temp1(A, B) {
    T = call1(A);
    Q = call2(B, T);
    return call3(Q);
}
```

```c
int temp2(A, E) {
    Z = call4(A, E);
    return call5(Z);
}
```

Notice that functions `call1`,...,`call5` are invoked in the same way as in `original_function`, but two new grouping functions are also introduced.

### 4.4 The framework

The proposed framework consists of three main components which are handled by independent program modules:

1. **Front-end**: performs source code to model **mapping**
2. **Simplifier**: carries out model **reduction**
3. **Comparator**: does model **comparison**

The **Front-end** creates a mathematical entity — which we call a model or an abstract view — from the source program to be examined. Subsequently, these views can be reduced in many ways by the **Simplifier**, creating different abstractions of the original model. Having the abstractions of two source programs, we use the **Comparator** to compare models on the same abstraction level and determine a similarity degree (a number between 0 and 1). As the abstraction becomes higher, the similarity of the abstract views is less and less indicative of the similarity of the original programs. Therefore we assign a factor (again a number between 0 and 1) to each abstraction level, with which we multiply the similarity degree obtained earlier.

Figure 4.1 shows the overview of the proposed framework. Here we start from two source programs `source A` and `source B`. The Front-end maps these sources to two models, `model A` and `model B`. Higher and higher abstractions of these models are produced by reductions, using the **Simplifier**. Finally, the models on the same abstraction levels are compared with each other.

In the following subsections we discuss in detail the main parts of the framework.

#### 4.4.1 Source code to model mapping

In general, the entity to which a program source is mapped can be chosen arbitrarily. For example, let us consider the size of the program source (e.g. in terms of characters used) as an abstract entity characterising the program, and consider the advantages and disadvantages of this choice. It is true that if we examine two entirely identical programs, then the comparison of their abstract views will signal match (the sizes of the programs will be the same). It also sounds feasible to consider the two program instances suspicious, if their size, in terms of characters, is exactly the same. However, if the programs are similar, but not identical, then the program size abstraction cannot give any hint on their similarity.
A further issue is that of simplifying transformations. When a program is characterised by its size, practically no further simplifications can be applied. The only, very weak option is to make further abstractions by rounding the size, e.g. using 1 kbyte instead of 1324 bytes.

Therefore, the abstract view must be more sophisticated (to allow diverse abstraction levels) and, more importantly, it must be possible to draw conclusions on the similarity of the programs from the similarity of the abstract views.

Therefore we suggest the use of directed, labelled graphs as the abstract views characterising the programs. Here the meaning of nodes, edges and their labels may vary from implementation to implementation. For example, the abstraction may be the program call graph, the data-flow graph of an execution, or — in case of object-oriented languages — the graph describing the object structure. The labels are used to describe the properties of the nodes and edges, e.g. to express that a node represents a built-in entity and not a user function.

Note that we suggest to ignore the labels in graph comparison, as we would like the similarity measure to focus on the graph structure. A further benefit of this is that it makes the comparison algorithms simpler and faster. However, the reduction steps do use the information stored in labels. This may result in somewhat strange effects: two graphs, that are considered isomorphic on one abstraction level, become non-isomorphic on the next level, provided the given reduction step uses the labels.

The graph representation is general enough to describe any kind of entity. As an extreme, even our first example, the program size abstraction, can be described as a labelled graph (with a single node whose label is the size).

### 4.4.2 Model reduction techniques — abstraction levels

One can envisage some kind of perfect mathematical models, that contain every bit of information present in the program source code. In this case we can be sure that, when two such perfect models are isomorphic, the corresponding program source code is the same. Of course, such a model is nothing else, but the source code itself in a different representation.

For any programming language and for any specific piece of source code, the lowest abstraction level, which we call level 0, could be considered to contain perfect models only. At first, one may think that the best one could do is to directly compare such perfect models. However, this may require a very sophisticated comparison algorithm, which is on one hand fast and easy to customise, and on the other hand
1. \( i = 1, Max = 0 \)
2. compare the two models on abstraction level \( i \), i.e. calculate \( S_i \)
3. calculate \( Max = \max(S_i * F_i, Max) \)
4. in case of isomorphism \( (S_i = 1) \), exit with the output value \( Max \)
5. if \( Max \geq F_{i+1} \), exit with value \( Max \), otherwise \( i = i + 1 \) and goto step 2

Figure 4.2: The algorithm for determining the similarity degree of two programs

resistant to the possible cheating methods mentioned in Section 4.3. Instead, we decided to follow a different approach using a series of views with increasing abstraction levels.

We thus propose to use several abstraction levels (as shown in Figure 4.1) and use relatively simple and fast comparison algorithms between models on the same level. Higher abstraction levels are built from lower ones (possibly utilising the labels in addition to graph structure) using certain transformations, called reduction steps. Our task is to transform the initial perfect models to ones which are more and more resistant to specific tricks, and which still represent the original program sources as much as possible.

Naturally, reduction steps are destructive operations: with every bit of dropped information we widen the gap between the perfect model and the model in question.\(^3\) Because of this, a perfect match (isomorphism for example) between two models on a high abstraction level “means less” than the same type of match on a lower level. To handle this, we assign a factor to each abstraction level in question, with which we multiply the similarity degree achieved on that level.

We define the similarity of two programs as

\[
\max_{1 \leq i \leq n} F_i S_i \tag{4.1}
\]

where \( n \) is the number of abstraction levels in the concrete implementation of the framework. \( F_i \) is the factor assigned to abstraction level \( i \) (a number between 0 and 1) and \( S_i \) is the actual similarity degree obtained on abstraction level \( i \) (also a number between 0 and 1). We require that \( F_{i+1} < F_i \) holds for any \( i \), i.e. the factors assigned to the abstraction levels form a strictly monotone decreasing sequence. However, we do not pose any restrictions on the values \( S_i \) and \( S_{i+1} \).

To determine the maximum, we may simply calculate expression (4.1). For example, let us assume we have two abstraction levels, level 1 and 2 with factors 1 and 0.9 respectively. If our models are assigned a 98% similarity on level 1 and are isomorphic on level 2, the algorithm calculates the values \( 0.98 * 1 = 0.98 \) and \( 1 * 0.9 = 0.9 \) respectively. The final result is the larger of these, namely 0.98.

We can optimise this naive algorithm in the following way. Whenever we detect that the maximum value we may obtain in the next abstraction level (which is \( F_{i+1} \) as \( S_{i+1} \leq 1 \) holds) is less or equal than the current maximum value \( Max \), we can stop. This is because the factors are decreasing, thus for every \( j = i + 1, \ldots, n \) it holds that \( F_j S_j \leq Max \). This trivially means that if we detect isomorphism between two models at abstraction level \( i \) we can immediately finish execution. When we stop, the final result (i.e. the similarity degree of the source programs in question) is the current maximum \( Max \). This algorithm is shown in Figure 4.2.

\[\] 4.4.3 Model comparison algorithms

In Section 4.4.1 we argued that directed, labelled graphs are good mathematical constructs for describing models of programs. Considering this, the concrete comparison algorithms are most likely related to graph theoretical algorithms.

\[\]

\(^3\)In theory we may end up in a point where every model becomes a singleton graph (a graph consisting of a single node): on this level every pair of models is isomorphic.
To calculate the code $C(T)$ of tree $T$:

1. determine the child subtrees of the root of $T$, $N_1, N_2, \ldots, N_k$
2. determine the codes for $N_1, N_2, \ldots, N_k$
3. sort the codes $C(N_1), C(N_2), \ldots, C(N_k)$ into ascending order using their binary values for ordering.

Assuming the concatenation of these produces a sequence $S$, return the sequence 150 as the code assigned to $T$.

Figure 4.3: The algorithm for calculating the code of a tree.

In general, our task is to define in what extent are two graphs similar to each other. Let us first consider the problem of graph isomorphism as an extreme case of graph similarity.

**Graph isomorphism**

The problem of graph isomorphism is the following. Given two graphs, $G$ and $H$, we look for bijection $f$ between the nodes of the graphs, so that $(x, y)$ is an edge in $G$ if and only if $(f(x), f(y))$ is an edge in $H$.

The graph isomorphism problem belongs to the class of NP problems, but we still do not know if it is NP-complete [3]. However, in special cases we know the complexity exactly or at least we can produce algorithms which run with acceptable speed. For example we know polynomial algorithms for planar graphs as well as for graphs where the maximum vertex degree is bounded [33].

In case of trees a more straightforward approach is applicable [71]. Namely, it is possible to construct a code in linear time for two trees $T_1$ and $T_2$, which fulfills the following two criteria:

1. if $T_1$ is isomorphic to $T_2$, then the code of $T_1$ equals to the code of $T_2$
2. if the code of $T_1$ equals to the code of $T_2$, then $T_1$ is isomorphic to $T_2$

Actually creating a tree code is nothing more than applying a geometrical transformation that maps a 2D tree to a one dimensional sequence of two characters. One can use the digits “0” and “1” or the parentheses “(” and “)” as the elements of the sequence, and accordingly the code of a leaf is “10” or, when parentheses are used, “()”. Let $T$ be the tree to be encoded and let $C(T)$ denote the code assigned to the tree $T$. Now, the recursive algorithm presented in Figure 4.3 assigns a binary number to any tree $T$.

Two examples of such encoding are given in Figure 4.4. We note that sometimes it is useful to apply a special notation for the leaves, to distinguish these from the code corresponding to other parts of the tree. We will use letter $L$ for this purpose. Accordingly, the codes in Figure 4.4 can be written as 1L0 and 1L1L00, respectively.

![Figure 4.4: Two examples for the coding scheme](image)

It is important that the algorithm in Figure 4.4 can also be used for DAGs (Directed Acyclic Graphs), i.e. directed graphs, not containing directed circles. In this case, in addition to a DAG, the algorithm requires
that a “root node” is specified, which serves as a starting point for the algorithm. An example of such a graph (the starting node is denoted by \( R \)) and its code can be seen in Figure 4.5.

Note that the code of a DAG can also be obtained by first transforming the DAG into a tree, and then taking the code of this tree. The transformation takes a vertex \( a \) with \( n > 1 \) incoming edges and replaces it by \( n \) new vertices, each with a single incoming edge (and each new vertex inherits all the outgoing edges of the original one). By repeating this transformation step we can eliminate all vertices with multiple incoming edges and thus obtain a tree from the DAG. The right hand side of Figure 4.5 shows the result of this transformation process, when applied to the DAG on the left hand side.

![Figure 4.5: A DAG and the corresponding tree with its code](image)

**Graph similarity**

Checking isomorphism is usually not enough by itself. The reason is that we cannot expect (at least on the lower abstraction levels) that the program graphs will be totally isomorphic, even with the most sophisticated source code to model mappings and reduction techniques. Actually we would like to detect if two graphs of hundreds of nodes (which are very typical for the programs we use) at a given abstraction level are nearly identical.

The general approach we use is to check how it is possible to transform one DAG code to another. For example, let us consider the following two sequences that correspond to the codes in Figure 4.4.

First sequence (\( F \)): 1L1LL0

Second sequence (\( S \)): 1LL0

The transformation steps we need to e.g. convert the first sequence into the second one can be described as follows: “remove 1 and then \( L \) from position 3 and 4 in sequence \( F \)”. Such transformation steps can be constructed by e.g. using algorithms solving the so called longest common subsequence (LCS) problem [57]. Let us denote by \( \Delta(A, B) \) the transformation steps between two arbitrary sequences \( A \) and \( B \). \( \Delta(A, B) \) is a set, containing pairs. The first part of such a pair can be 0, 1 or \( L \), the element to be added or removed. Each of these elements is preceded by either a plus(+) or a minus(−) sign, corresponding to element addition and removal. The second part of a pair is an integer, describing the position the specific transformation step should be applied at. In our case, \( \Delta(F, S) \) is the set \{((-1, 3), (-L, 4))\}.

To determine the similarity degree of two arbitrary codes we assign penalties to the specific transformation steps. For example, we may say that the removal of a leaf (i.e. a \(-L\) in the transformation set) reduces the similarity by a certain amount, let us say by 0.01. Using the penalties we calculate \( \Omega(A, B) \), the discrepancy function describing to what extent codes \( A \) and \( B \) are different:

\[
\Omega(A, B) = \min \left( 1 - \sum_{E \in \Delta(A, B)} P(E) \right)
\]  

(4.2)

Here \( P \) is the penalty function that assigns a value to a given type of transformation step. Using \( \Omega(A, B) \) we define the similarity degree of graphs \( A \) and \( B \) as:

\[
1 - \Omega(A, B)
\]  

(4.3)
Let us note that what we actually calculate here is a variant of the so-called Levenshtein and edit distances. The Levenshtein distance [80] between two strings is the minimal number of operations needed to transform one string into the other. By operation we mean an insertion, deletion, or substitution of a single character. The edit distance [25] is a generalisation of the Levenshtein distance in that the operations have costs assigned to them, similar to the costs we have defined above.

**Distinct paths in the graphs**

Unfortunately, in a special case very similar graphs are considered to be far from each other according to the similarity degree introduced above. The reason for this is the way how DAG codes are built. We have seen in Figure 4.5 that the code corresponding to node $C$ appears in the code of the whole DAG twice (i.e. we have two $L$ characters in the code, although the original DAG has only one leaf). In general, for any DAG $G$, the code of a node $v$ appears in the code of $G$ exactly $m$ times, where $m$ equals to the number of distinct paths from the root to $v$.

Let us assume that the DAGs corresponding to programs $A$ and $B$ differ only slightly in a single node, which is, however, accessible from the root along many distinct paths. Because of the reasons outlined above, the DAG codes corresponding to $A$ and $B$ will differ significantly.

We suggest two ways to overcome this shortcoming, both of which are used in our prototype system described in the next section. First, we suggest to use reduction steps which decrease the number of distinct paths from the root, thus making the graphs more “tree like”. For example, filtering multiple edges in a graph (cf. Section 4.5.2) reduces the number of distinct paths significantly.

We now describe how to compute the value of $\Omega'(A, B)$ for arbitrary two graphs $A$ and $B$. We assume that a popular node $N$ in $A$ can be associated with its counterpart $M$ in $B$, using some heuristics: for example, we can pair those popular nodes whose number of incoming edges and number of arguments are the closest. If such a pairing is not possible (if at most one of the graphs contains popular nodes) then $\Omega'(A, B)$ simply equals to $\Omega(A, B)$. If a pairing is possible then we first apply our algorithm recursively to the subgraphs rooted at $N$ and $M$, i.e. we calculate the value $L = \Omega'(N, M)$. Next, we create DAGs $A'$ and $B'$ from the original ones by replacing $N$ and $M$ by single nodes, having no outgoing edges. The modified discrepancy is then calculated recursively as

$$\Omega'(A, B) = \Omega'(A', B') + L$$

The algorithm introduced above is summarised in Figure 4.6. In our implementation we offer the user a choice of the discrepancy function ($\Omega$ or $\Omega'$) through the graphical user interface (see Section 4.5.5).

---

As a second solution we suggest to introduce a slight modification of the similarity degree as defined in (4.3). This modification relies on identifying nodes with numerous incoming edges (let us call these nodes popular)\(^4\). Using structural decomposition we suggest to calculate a variant of the discrepancy function $\Omega$ specified in Equation 4.2, called $\Omega'$. The final similarity degree will thus be $1 - \Omega'(A, B)$.

We now describe how to compute the value of $\Omega'(A, B)$ for arbitrary two graphs $A$ and $B$. We assume that a popular node $N$ in $A$ can be associated with its counterpart $M$ in $B$, using some heuristics: for example, we can pair those popular nodes whose number of incoming edges and number of arguments are the closest. If such a pairing is not possible (if at most one of the graphs contains popular nodes) then $\Omega'(A, B)$ simply equals to $\Omega(A, B)$. If a pairing is possible then we first apply our algorithm recursively to the subgraphs rooted at $N$ and $M$, i.e. we calculate the value $L = \Omega'(N, M)$. Next, we create DAGs $A'$ and $B'$ from the original ones by replacing $N$ and $M$ by single nodes, having no outgoing edges. The modified discrepancy is then calculated recursively as

$$\Omega'(A, B) = \Omega'(A', B') + L$$

The algorithm introduced above is summarised in Figure 4.6. In our implementation we offer the user a choice of the discrepancy function ($\Omega$ or $\Omega'$) through the graphical user interface (see Section 4.5.5).

---

\(^4\)In our implementation (see Section 4.5) nodes with at least 10 incoming edges are considered popular.
4.5 The prototype implementation

In the following we present our test implementation of the framework, the Match plagiarism detection tool. The current version of Match supports two Front-ends, for Prolog and SML, as we teach these two languages as part of a Declarative Programming course, and major assignments must be written in these languages. Here we describe the Prolog Front-end only, more about the SML interface can be read in [52].

In the following we discuss the implementation details of the relevant parts of the comparison framework (Front-end, Simplifier and Comparator), then we describe the graphical user interface of the system.

4.5.1 Mapping source code to a model

We chose call graphs as the models of Prolog programs. A call graph is a graph where the nodes correspond to the Prolog predicates and the edges to the calls. If in the body of predicate \(A\) there is a call to predicate \(B\) then an edge between the nodes \(A\) and \(B\) is present in the graph. When there are multiple calls, multiple edges are present. We decided to exclude the built-in predicates (such as \(is/2\)) from the graph, because they do very elementary tasks and would increase the graph size considerably, without increase in the precision of the model. The graph includes, however, the library predicates and also reflects implicit meta calls, made by using \(findall/3\), for example.

Furthermore we made some simplifications to our model: we remove from the call graph the self-loops, which correspond to recursive calls. This is because explicit recursion is so common is Prolog that for us it does not contain valuable information. We also remove back edges (i.e. edges which point to an already visited node during a depth-first search) in order to avoid cyclic graphs, so that we can work with DAGs instead of general graphs. Although this means that simple reordering tricks can change the resulting graph (as they change the order in which a depth-first search visits the nodes) we do not consider this a problem. This is because in our model, circles actually correspond to so called mutual recursion (for example when \(A\) calls \(B\) and vice versa). Our experience, however, is that mutual recursion is very rarely used by students and so neglecting it does not effect the final similarity measure in a significant way.

Finally, those predicates to which there was no reference in the source code are not included in the graph, i.e. the call graph consists of a single component.

Call graphs are well suited for Prolog programs. This is because the only control structure of Prolog is the predicate invocation, it lacks \(while\), \(for\), \(goto\) or any other “usual” imperative control elements.

The call graph is built from the program source code by using static source code analysis. For this we slightly modified the \texttt{xref} package of SICStus Prolog.

4.5.2 Model reduction techniques

For the comparison of Prolog programs we have defined four reduction steps, which are applied in succession. This means that the comparison can be performed on five abstraction levels. To each reduction step we have assigned a similarity factor as introduced in Figure 4.1. These factors were chosen empirically. The actual reduction steps and their similarity factors are the following:

1. non-called predicates: we remove those predicates which were not called during a test call. The test call is provided by the tester. Test calls are usually simple tests which are easily solvable by the programs. The assigned similarity factor is 0.95.

2. library or dynamic predicates: here we omit the library and dynamic predicates from the call graphs. The similarity factor is 0.9.

3. multiple edges: we remove multiple edges from the call graph and only keep one edge between any given two predicates. This helps handling the popular node problem presented in Section 4.4.3. We set the similarity factor to 0.8.

4. topological isomorphism: we remove those vertices from the call graphs which have a degree of 2 (one incoming and one outgoing edge). This helps to detect the call-tunneling trick. The similarity factor here is 0.7.
Actually the user of the plagiarism detection system can decide to skip some of the reductions steps (cf. Section 4.5.5), which results in less than 5 abstraction levels. In this case the factors can be different from the ones shown above, as the value of a factor is usually set in a “context sensitive” way, considering what other reduction steps have been done previously. For example, in our concrete implementation, if we use the first, second and fourth reduction step, but we do not filter multiple edges, the factor for the highest abstraction level (the one corresponding to step four above) is set to 0.85. Thus step 4 when applied after steps 1 and 2 is considered to be slightly less “destructive” than step 3 in the same context (the corresponding factors being 0.85 and 0.8, respectively). Our experience is that using step 3 and 4 simultaneously has significant cumulative effect, justifying the similarity factor of 0.7, when all the above reduction steps are applied.

4.5.3 Model comparison algorithms

In our system the comparison of models is based on the coding technique and the similarity degree introduced in Section 4.4.3. We have chosen this approach because we found that comparing codes often gives a good intuitive characterisation of the similarity of the programs.

For example, if the codes match the corresponding models are trivially the same. If one of the codes contains the other as its subsequence, then it can be suspected that one student got the other’s program and added some new structure to it.

Call grouping can also be detected from the codes\(^5\). For example if predicate \(P\) calls \(T\) which calls four other predicates, the corresponding code will be \((L (L L L L))\), where parentheses represent binary values and \(L\) represents a leaf as described in Section 4.4.3. If we apply call grouping, for example \(T\) will call \(Q\) and \(W\), each of which will call two other predicates, then the code takes the form \((L ((L L) (L L))))\). Here the second, third, fourth and fifth parenthesis has to be removed in order to get the original code.

We use the widely available UNIX \texttt{diff} program to actually enumerate the differences between the codes \(A\) and \(B\), i.e. to calculate \(\Delta(A,B)\). The \texttt{diff} program uses a variation of the LCS algorithm (cf. Section 4.4.3). The way we make use of \texttt{diff} is the following. First we make two files corresponding to the two codes we would like to compare. The way we create the content of such a file is the following: each \((,)\) and \(L\) in the tree code is put on a separate line. For example the file corresponding to the code \((LL)\) shown in Figure 4.4 will contain four lines:

\[
\begin{align*}
&\text{\} \\
&\text{L} \\
&\text{L} \\
&\text{)}
\end{align*}
\]

Next, we let the UNIX \texttt{diff} utility calculate how these files can be made equal, i.e. to produce the instructions on which leaves and nodes should be added or removed to make call graphs \(A\) and \(B\) isomorphic. Actually we always try to modify the bigger graph (i.e. the graph with longer code) and check what transformations we can use to obtain the smaller one.

By analysing the information given by the \texttt{diff} utility we assign a similarity degree to the pair of codes. As we described in Section 4.4.3, we start with degree 1 and for each difference we subtract a “penalty” fraction, which reflects how much we should “punish” the given modifications of the code sequences. This corresponds to calculating equation (4.3) in Section 4.4.3. We found that the penalties shown in Table 4.1 are very usable\(^6\):

Accordingly, if we need to remove one leaf from our bigger call graph to make it identical to the smaller one, then the similarity degree is 0.99. As one can see, addition is always penalised more than removal. This is because of our experience that cheating students usually try to copy and modify the work of a fellow student while keeping the original parts intact. This way the original program will be part of the resulting (bigger) program. So, our assumption is that in case of plagiarism the bigger graph can be reduced to the smaller one by applying node and edge removals.

---

5We will use parenthesis in the codes below, instead of binary digits, as this makes call nesting more apparent.

6Note that the addition and removal of a node actually corresponds to two differences, one for the opening and the other for the closing parenthesis.
According to this assumption, we actually offer to use the \texttt{diff} program in two different modes. The mode named \texttt{sdiff} (simple \texttt{diff}) means that we only consider those transformations that require only node or edge removals from the bigger graph. Otherwise we conclude that no plagiarism happened. In the other mode, called \texttt{fdiff} (full \texttt{diff}), we make no such assumption. This results in more false positive results, but it also increases recall significantly (see Section 4.6).

Having explained the concrete implementation, we reiterate the issue of abstraction levels. Let us consider two graphs which differ only in the multiplicity of the edges. In the absence of abstraction levels, using \texttt{diff} alone, we could easily get a similarity degree of 0, provided there is a sufficient number of multiple edges in the graph. However, when the multiple edge removal abstraction is applied we get a similarity of 0.9, which may be more appropriate. This shows that the introduction of abstraction levels is a useful extension, in addition to the \texttt{diff} algorithm.

We have also made a further improvement in the calculation of the similarity degrees, as discussed in the following subsection.

### 4.5.4 Generating mappings between predicates

Although by calculating the similarity degree of the source programs and presenting the user the most promising pairs we have already fulfilled our original goal, it greatly helps the user of the system if we present some kind of a “proof” of cheating as well. We produce such a “proof” in the form of a mapping between the predicates of the two programs. In this mapping a predicate of one program is paired with the predicate of the other program which is most similar to it. This is a very useful guide to the user when she verifies the results manually.

These mappings are generated by a systematic deterministic traversal of the codes in question. We start from the nodes corresponding to the root predicate (i.e. a predicate which is the entry point of all the student programs). These nodes are paired with each other. Then we visit the neighbours of the starting nodes and pair them using their codes. We continue this algorithm recursively. Whenever there is ambiguity, e.g. we are examining two nodes with multiple neighbours having the same codes, we use a heuristic: those nodes will be paired whose number of arguments differ the least. Note that the mappings are actually derived from the models belonging to the abstraction level where the maximal similarity degree was found. An example mapping is shown in Figure 4.8.

As mentioned above, the “quality” of the mapping is also taken into consideration when calculating the similarity degree. In the current implementation we actually decrease the similarity degree of the programs by 0.005 for each pair of predicates mapped to each other, which have different arities.

### 4.5.5 The graphical user interface

Match offers a graphical user interface (GUI) where the user can customise the parameters of the comparison and can view the results. A screenshot of the main window can be seen in Figure 4.7 (it shows the state of the system right after a successful execution). On the top of the window we find four buttons. The

![Table 4.1: Penalties used by the Match system.](image)

<table>
<thead>
<tr>
<th>Type of modification</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>removal of a leaf</td>
<td>0.01</td>
</tr>
<tr>
<td>addition of a leaf</td>
<td>0.03</td>
</tr>
<tr>
<td>removal of a node</td>
<td>0.02</td>
</tr>
<tr>
<td>addition of a node</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Note that we changed the name of the students because of privacy issues.
first one called “Make info” invokes the Front-end, i.e. it creates the models from the source codes. This practically means that Match searches for source programs in the given directory and for each source code it creates a special file containing the labelled call graph.

These files are loaded by the second button named “Load info”. In the bottom of the window we can see that in this specific example we loaded 32 such graphs.

The third button called “Analyse” starts the comparison process based on the parameters the user specified by using the check boxes located in the “options” and “advanced” areas. These options basically tell Match what kind of reduction steps it should apply, whether it should use the \texttt{diff} algorithm\footnote{If this option is disabled, then isomorphism is used instead of similarity, i.e. the similarity degree of the models is considered to be a binary value: 1 means that the models are isomorphic, 0 means they are not.} and what is the similarity threshold (i.e. only hits with similarity degree greater than the threshold will be presented). In the example shown in Figure 4.7 we selected all the reduction steps, asked Match to use \texttt{diff}, take special care of the popular nodes (called famous in the GUI) and set the threshold to 60%.

When we use the \texttt{diff} algorithm, we can also set the penalties (in units of 0.01) the Match program should
use to determine the similarity of the two graphs. In this example the penalties are set to the ones described in Section 4.5.3.

After a successful execution the suspicious pairs of programs are shown in the middle of the screen under the title “Results”. In Figure 4.7 we have 15 such pairs. If we select one of these pairs, Match displays the predicate mapping between the two source programs. Namely, we can see which predicate in one program matches which predicate in the other, as shown in Figure 4.8. In this concrete mapping we can see that, for example, predicate kul corresponds to predicate kulonb, both of them having 3 arguments. We can also see that the similarity of these programs is calculated to be 68.5%, and that this was found on abstraction level 4. The next line, “Type:”, indicates that even on this abstraction level the codes were only "similar", i.e. non-isomorphic.

4.6 Evaluation

Below we first reiterate our goals presented in Section 4.3 and examine how they are fulfilled by the Match system. Next, we present real life execution results showing that the framework convincingly detects plagiarism in student programs.

Table 4.2 summarises the student tricks we have described in Section 4.3 and for each gives a brief explanation of how the given trick was handled in the implementation of the plagiarism detection framework.

<table>
<thead>
<tr>
<th>Student trick</th>
<th>Preemptive measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>changing the names of the identifiers</td>
<td>we do not store names in our models</td>
</tr>
<tr>
<td>changing the arity of the functions</td>
<td>arity is not used in model comparison</td>
</tr>
<tr>
<td>splitting the program into several modules</td>
<td>module boundaries are not taken into account</td>
</tr>
<tr>
<td>reordering the function definitions</td>
<td>the call graph is not affected</td>
</tr>
<tr>
<td>reordering the statements within a function</td>
<td>the call graph is not affected</td>
</tr>
<tr>
<td>putting useless functions in the program</td>
<td>using the “non-called predicates” reduction step</td>
</tr>
<tr>
<td>putting useless calls in a function</td>
<td>using the “non-called predicates” reduction step</td>
</tr>
<tr>
<td>call-tunneling</td>
<td>using the “topological isomorphism” reduction step</td>
</tr>
<tr>
<td>call-grouping</td>
<td>using the diff algorithm</td>
</tr>
</tbody>
</table>

Table 4.2: Our goals and the way how they are achieved.

We now proceed to discuss the performance evaluation of the Match system. We were lucky enough to have abundant amount of Prolog source programs to test the prototype on. Moreover, several students were kind enough to provide us with some hints on what cases were they cheating (after they completed the course and were promised full amnesty). So we had the minimal expectation that the Match system should at least mark those assignments as matched pairs.

The 73 source programs\(^9\) were evaluated against 4 different similarity thresholds. For example, the

\(^9\)We had actually 92 submitted Prolog homework, but from these we excluded those programs that either do not compile or do not solve the required number of test cases.
60% threshold means that our system shows pairs of source codes which have similarity degree at least 60% percent. For every threshold, the system was run with 18 different parameter variations. These include the most useful settings in practical cases. These variations are based on the following 6 basic variants:

- variant B: all options are disabled (base case)
- variant N: filtering non called predicates
- variant NL: N + filtering library/dynamics predicates
- variant NLM: NL + filtering multiple edges
- variant NLT: NL + topological isomorphisms
- variant NLMT: NLM + topological isomorphisms

In the first six cases we do not use \textit{diff}, i.e. we only consider graph isomorphism between the models at different abstraction levels. The next twelve variations are obtained by applying \textit{diff} in two different modes, simple and full (cf. Section 4.5.3).

For every setting we measured the run time, the number of hits and the ratio of hits and the relevant hits (precision). To determine the relevant hits we examined the program pairs manually, and decided if the given case should be considered plagiarism or not. We also made a serious manual effort to check if there were any cases of plagiarism not found by Match. We concluded that the 13 pairs discovered by the most complex run of Match were the only cases where one program code was derived from the other one.

Assuming that the number of all hits is 13, we calculated the so called \textit{recall} which is the ratio of the number of relevant hits returned by the program and the number of all hits.\textsuperscript{10} The results for threshold 60% are listed in Table 4.3. The test were run under Linux on Intel Celeron 450Mhz processor with 128 MByte of RAM.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Execution time</th>
<th>Compared pairs</th>
<th>Hits</th>
<th>Relevant hits</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>15.40s</td>
<td>2628</td>
<td>6</td>
<td>6</td>
<td>46%</td>
<td>100%</td>
</tr>
<tr>
<td>N</td>
<td>20.33s</td>
<td>2628</td>
<td>6</td>
<td>6</td>
<td>46%</td>
<td>100%</td>
</tr>
<tr>
<td>NL</td>
<td>30.11s</td>
<td>2628</td>
<td>6</td>
<td>6</td>
<td>46%</td>
<td>100%</td>
</tr>
<tr>
<td>NLM</td>
<td>31.50s</td>
<td>2628</td>
<td>6</td>
<td>6</td>
<td>46%</td>
<td>100%</td>
</tr>
<tr>
<td>NLT</td>
<td>31.28s</td>
<td>2628</td>
<td>6</td>
<td>6</td>
<td>46%</td>
<td>100%</td>
</tr>
<tr>
<td>NLMT</td>
<td>33.40s</td>
<td>2628</td>
<td>16</td>
<td>8</td>
<td>61%</td>
<td>50%</td>
</tr>
<tr>
<td>B + sdiff</td>
<td>60.13s</td>
<td>2628</td>
<td>8</td>
<td>6</td>
<td>46%</td>
<td>75%</td>
</tr>
<tr>
<td>N + sdiff</td>
<td>78.07s</td>
<td>2628</td>
<td>8</td>
<td>6</td>
<td>46%</td>
<td>75%</td>
</tr>
<tr>
<td>NL + sdiff</td>
<td>124.41s</td>
<td>2628</td>
<td>17</td>
<td>8</td>
<td>61%</td>
<td>47%</td>
</tr>
<tr>
<td>NLM + sdiff</td>
<td>179.78s</td>
<td>2628</td>
<td>22</td>
<td>8</td>
<td>61%</td>
<td>36%</td>
</tr>
<tr>
<td>NLT + sdiff</td>
<td>162.34s</td>
<td>2628</td>
<td>17</td>
<td>8</td>
<td>61%</td>
<td>47%</td>
</tr>
<tr>
<td>NLMT + sdiff</td>
<td>364.04s</td>
<td>2628</td>
<td>32</td>
<td>10</td>
<td>76%</td>
<td>31%</td>
</tr>
<tr>
<td>B + fdiff</td>
<td>98.2s</td>
<td>2628</td>
<td>48</td>
<td>10</td>
<td>76%</td>
<td>21%</td>
</tr>
<tr>
<td>N + fdiff</td>
<td>122.02s</td>
<td>2628</td>
<td>48</td>
<td>10</td>
<td>76%</td>
<td>21%</td>
</tr>
<tr>
<td>NL + fdiff</td>
<td>180.78s</td>
<td>2628</td>
<td>65</td>
<td>11</td>
<td>84%</td>
<td>17%</td>
</tr>
<tr>
<td>NLM + fdiff</td>
<td>295.6s</td>
<td>2628</td>
<td>70</td>
<td>11</td>
<td>84%</td>
<td>16%</td>
</tr>
<tr>
<td>NLT + fdiff</td>
<td>232.60s</td>
<td>2628</td>
<td>65</td>
<td>11</td>
<td>84%</td>
<td>17%</td>
</tr>
<tr>
<td>NLMT + fdiff</td>
<td>414.43s</td>
<td>2628</td>
<td>80</td>
<td>13</td>
<td>100%</td>
<td>16%</td>
</tr>
</tbody>
</table>

Table 4.3: Match results for the threshold of 60%.

\textsuperscript{10}Precision and recall are measures widely used for evaluation information retrieval techniques, see e.g. in [53].
From our tests we can draw several conclusions. First, we found that students do not often use sophisticated tricks. This can be seen from the fact that adding more reduction steps does not significantly improve the effectiveness of Match: most of the cheaters are caught already at the lower levels. At the same time further reduction steps do result in new hits, so higher abstraction levels are by no means useless. For example, 80% percent of the cheaters found in the second block were already uncovered after two reduction steps (variant NL). These included the programs of a pair of students who claimed they worked on the modifications for more than 5 hours, and in spite of this, their similarity degree was nearly 90%.

We can also see that although precision drops back significantly as we use more and more abstraction levels, the results are still acceptable. In the worst case (here the precision is 16%) one in six pairs of codes is a proper hit among 70 suspicious pairs. Although this requires some effort from the person verifying the results of Match, the amount of manual work is still almost two magnitudes less than that required when the plagiarism detection framework is not used (over 2500 cases).

We can also conclude that both the abstraction mechanism and the similarity degree calculated in (4.3) are needed. There were cases when the plagiarism was detected with a high degree of similarity due to the fact that only minor differences were found on abstraction level 1, for example. Without considering graph similarity, we would have needed to use more reduction steps to make these models isomorphic, resulting in a smaller similarity degree. This shows how useful the technique introduced in (4.3) can be. However, the opposite situation also occurred. Namely, we could find programs (relevant hits) which were isomorphic on a relatively high abstraction level, but calculating (4.3) on the previous level gave much lower similarity. This shows the significance of using reduction steps.

Our approach is very fast. For 73 programs, checking of all the pairs took between 15.4s (no diff and no abstraction levels) and 414s (when all of the reduction steps and the full diff algorithm were applied).

4.7 Future work

Our future plans include the integration of the most promising statistical and/or lexicographic approaches in the framework. This way we can use hybrid comparison techniques which we hope can be more efficient than the pure structural approach in some cases.

We would also like to develop Front-ends for further programming languages. In case of procedural languages, such as C, Pascal or several script languages, the control structures are the main bearers of information. When dealing with object-oriented languages, C++ or Java for example, the class hierarchies should also be properly represented in the models.

The heuristics used by the Match system can also be improved. These include the heuristics used when pairing popular nodes and predicates between two models. Furthermore, the user interface lacks some functionality. For example, it is often the case that a larger group of students submit similar assignments. In such a case Match returns all pairs within the group as suspicious. It would help the person verifying the results, if the group of cheaters was identified, as a whole.

4.8 Conclusion

In this chapter we have presented a plagiarism detection framework which is capable of calculating a similarity degree for a pair of program sources. The framework uses directed, labelled graphs to represent the structural information extracted from the programs. Instead of using sophisticated comparison algorithms our approach combines the use of relatively simple comparison techniques together with simplifying graph transformations, called reduction steps.

We have presented the three main components of the generic framework: the Front-end which converts programs to graphs, the Simplifier, which carries out the reduction steps and the Comparator, which calculates a similarity degree for the graphs. We have described the implementation of the framework, the Match system, which has been successfully used to detect plagiarism in homework assignments for years. We have also presented a detailed performance evaluation of the system.

We believe that the novel architecture of our approach, based on simplifying graph transformations and straightforward comparison algorithms, has proved to be a viable technology for plagiarism detection in source programs.
Chapter 5

Logic based management of documents

This chapter presents the the SREngine framework which manages a pool of generic objects together with their properties and supports reasoning on these. The main idea of the system is to infer new properties about the objects, using their existing properties and a set of user defined rules. These rules are given in a special extensible logic programming language introduced in the chapter, providing intuitive syntax and adequate expressive power.

In a typical scenario the SREngine is used to classify documents, i.e. to attach type information to them. For this we usually need to analyse the textual content of the documents which is done by dedicated information extractor modules. We argue that the rule based approach we apply is better suited for this task than standard ontology based approaches.

The SREngine realises a bottom-up reasoner fully implemented in Prolog. The system is intended to be used in real-life applications, providing robust implementation and web-service based interfaces for standardised information exchange.

5.1 Introduction and goals

The Sense/Net Rule Engine (SREngine) is a generic rule based inference system implemented in Prolog. The SREngine is primarily designed to cooperate with host systems managing documents together with their properties (so called document stores). These include Web portals, content management systems and certain business applications dealing with documents or similar entities. The basic motivation of this research and development work is to add transparent reasoning capabilities to such a system, which we will simply refer to as the host system in the rest of the chapter.

Our high-level goal is to use logic based techniques to achieve more intelligent behaviour of queries involving documents. This is done by providing a knowledge representation formalism for storing background knowledge, and using it to infer new properties within the document store, i.e. provide new pieces of meta-information about documents. These can then be utilised during query execution, a task delegated to the host system.

As an additional benefit, user interfaces provided by the host system can also become more intelligent, for similar reasons. For example, the SREngine can be used (1) to infer that some documents are related to each other and (2) to represent this knowledge in the document store. Now, whenever the user navigates to one of these documents, the related documents can also be presented to her creating a much richer user experience. Note that the definition of „related to” depends on the concrete application. For example it may link documents that are similar to each other based on some similarity measure or simply expresses the fact that the related documents have the same author, type etc.

The chapter is structured as follows. First, in Section 5.2, we examine related work. In Section 5.3 we give a general introduction to the SREngine system describing the main components and their interactions. In the next section we discuss our rule language called SRLang, introducing the syntactic constructs and discussing the modelling decisions we have applied. In Section 5.5 we present the execution process of the rules and the implementation details of the SREngine system. Section 5.6 evaluates the SREngine system. Finally, we conclude with a summary of our results and the discussion of the future work.
5.2 Related work

Document classification in general is a huge field in computer science. A large variety of approaches have been applied in the past few decades involving machine learning, neural networks, pattern recognition, natural language processing and other AI related fields [107, 70]. There are also numerous approaches which rely on some form of rules [21, 51]. The common properties of these systems is that they aim to cluster a given collection of documents into groups that have similar contents.

Graphs are closely related to document stores as these can easily be represented in an appropriate graph description language, such as the RDF framework [10] used by the Semantic Web community. For RDF, several rule languages have been proposed such as the TRIPLE [121], the SWRL [61] or the RuleML [119], all of which are expressive and are based on Horn logic.

Ontology description formalisms, especially Description Logic (DL) [4] and its syntactic variants, such as OWL [8], can also be used for document classification. Here the idea is that (1) we define the document categories by providing appropriate DL terminology axioms, (2) then we perform instance retrieval over an ABox where the instances correspond to the actual documents of the given system. Details of such an approach are described, for example, in [18]. A common problem with this approach is that in case of huge amounts of underlying data the performance may not be adequate compared to rule based systems (although recently there was serious progress here, see Chapter 2 of the present dissertation). Another problem is that the Open World Assumption implied by the usage of Description Logic does not seem to fit well with the expectations of the users of a document management system.

Several approaches have been proposed which focus on the combination of querying and navigation and use logic for inferring new properties, e.g. [36]. Deductive databases [19] are also strongly related to our approach as the syntax and execution of Datalog, in particular, resembles our rule language to a great extent. There have also been serious work done on how to apply logic to databases and information systems [81]. The rule language of the SREngine system has the important property that it is extensible via Web Service calls (cf. Section 5.4.3). In this aspect it is related to other frameworks for Logic Programming extended with generalised external function calls such as [31, 30].

There are common metadata vocabularies for document annotations which fit well to our approach. For example, we encourage the creator of the rules to use elements from the Dublin Core element set [27] (e.g. to use property dc:creator instead of something like hasAuthor) as, in our specific case, the host system provides several convenience facilities of Dublin Core. Note, however, that in the rest of the chapter we do not use Dublin Core in our examples.

We believe that the SREngine system primarily differs from the above mentioned approaches because it tries to encapsulate several ideas in a single document management framework. For more detailed comparison see Section 5.6.

5.3 Overview of the SREngine system

In this section we give a general overview of the SREngine system. First we introduce the notion of document store by describing its properties and its content. Next, we discuss the various usage scenarios of the SREngine system. Finally, we introduce the main components: we discuss the general architecture of the system and describe the communication interfaces of SREngine.

5.3.1 The document store

At an abstract level, a document store is considered to be a graph, where the nodes represent document-like entities or values. A document-like entity is an object representing a document, the actual content of a document, a directory, a document category, or any other object related to documents. If this does not cause confusion, we will simply refer to document-like entities as documents.

The edges in a document store, called properties, run between documents or between documents and values. In the first case, properties represent relations between documents. For example, we can describe that document A is the draft version of document B. In the second case properties describe the attributes of
the documents, such as the title, the authors, etc. We note that this model actually corresponds to existing knowledge representation formalisms, such as RDF [10].

Some properties are special. The property `has_binary` is used to connect a document with its content. The distinction between a document and its content is needed because the latter is very often given in a non-textual, binary representation (e.g., in case of Microsoft Word documents). The transitive property `has_child` is used to describe the hierarchical inclusion relation between the nodes, i.e., to express the fact that a document is located in a directory or that a document category is the subcategory of another. We assume that every document is reachable from a root document via one or more `hasChild` links, and that these links form a tree. This provides the basis for path expressions (see Section 5.4).

### 5.3.2 Usage scenarios and methodology

The SREngine system works with rules, written in a custom designed logic language, which describes how to infer new properties from existing ones, using information retrieved from the actual content of the documents. For obtaining the latter we use `Information extractor` modules. These software components are specialised to extract certain kinds of information from the documents.

In the simplest scenario, the SREngine system uses rules most of which simply add properties extracted by the Information Extractor modules. In this scenario the system actually fills in those pieces of meta-information that the users of the host system have not supplied for some reason (e.g., because of lack of time, migration of the already existing document structure, etc.). Typically, this means that the SREngine system is used to fill in some document properties, such as the title, the authors, etc. The most important of these are the properties specifying the document classification, e.g., whether it is a contract, a note, etc.

In another scenario, rules are primarily used to infer new, complex relations between the nodes of the document store using the available meta-information. Such relationships often represent information that one cannot expect the users to specify. For example, one can create a rule that counts how many other documents refer to the given document. This can be useful in certain situations, for example, when one would like to rank the documents based on their importance.

Practically, of course, the SREngine is used in the mixture of the modes described above. The main objective of the system is to derive new information which helps us to achieve our generic goals, i.e., to answer user queries more intelligently and to provide more user friendly interfaces.

In the context of the host system we distinguish between system experts, administrators and end-users. Experts work for the host company (i.e., the creators of the host system) and are responsible for deploying the host system at the client site. Administrators have the right to manage the document store of the system: they can add or remove documents, attach meta-data to them, create queries involving documents, etc. Finally, end-users simply use the portal application for their everyday job.

By extending the host system with the SREngine module the tasks of the administrators now also include the creation of the rules. They are the experts who should be trained to understand the rule language and the corresponding execution mechanism. From the point of the end-users the reasoning process is fully transparent: for them it seems as if all the inferred properties were extracted from the documents directly or specified manually.

![Diagram](image.png)

**Figure 5.1:** Inputs and outputs of the SREngine system
5.3.3 Architecture

Figure 5.1 shows the SREngine as a black box focusing on its I/O behaviour. The system has two inputs: (1) the content of the document store and (2) a set of rules written in the SRLang language (see Section 5.4). Using these, the system produces instructions on how to change the content of the document store.

The execution is batch-like, i.e. the SREngine is supposed to be executed from time to time, preferably at a time when the host system is offline, i.e. its document store does not change during the reasoning process (see Section 5.7).

Figure 5.2 shows the detailed architecture of the SREngine system together with an outline of the Document Store. The boundary of the SREngine is indicated by a dashed line. Accordingly, the SREngine consists of two main parts: the Knowledge Base and the Reasoner.

The Knowledge Base stores the rules and the content of (the relevant part of) the Document Store. This part excludes the binaries, i.e. the SREngine system only works with the meta-information of the documents and not with the documents themselves. This creates the possibility to use existing infrastructures of metadata storage such as Jena [56] or Sesame [1]. The two parts of the Knowledge base are called (local) Rule Store and Local Document Store, respectively. The Reasoner works on the information available in the Knowledge Base and on the data the Information Extractors provide. The results are written back to the Knowledge Base (note arrow (4) from the Reasoner to the Knowledge Base).

Finally, the role of the Information Extractor components is to extract useful information, such as the name of the author, from the binaries of the documents in the document store. This usually involves using more or less sophisticated data extraction techniques. The extracted information is used by the Reasoner. Although the Information Extractors are usually modules developed specifically for the SREngine system, Figure 5.2 indicates that they belong to the host system. The reason for this is that the Information Extractors must communicate with the host system in a very efficient way as they work with huge amounts of data corresponding to the binaries (cf. arrow (6) in Figure 5.2). Thus, this component is expected to be part
of the host system and hence from the point of the SREngine system they are considered black-boxes.

Once the reasoning process is completed, the content of the Local document store is synchronised with that of the document store maintained by the host system, as indicated by arrow (2) between these components in Figure 5.2. Synchronisation is done by transmitting an appropriate set of modifications: a new property graph containing those nodes and properties that were involved in the reasoning process. The responsibility of the SREngine ends here: the host system takes care of the remaining part of the process, possibly involving even human control.

5.3.4 Interfaces

The SREngine system uses several kinds of interfaces to communicate with the host system. In Figure 5.2 we have three such interfaces (1)-(3). Interface (1) makes it possible to populate the content of the Rule Store, while (2) does the same for the Local Document Store. Interface (3) defines the communication between the Reasoner and the Information Extractor component. As the Information Extractor is part of the host system its communication interface (6) is not covered here. In the following we discuss these interfaces in more detail.

Rule Store interface

The SREngine system supports two formats for reading SRLang rules: it is capable of processing rules given as text files as well as reading rules in XML format (see Section 5.4). Technically, the rules are read either from a file or via SOAP based Web Service calls. In the latter case, the host system is expected to provide the appropriate Web Service which the SREngine system can invoke.

The interface specifies the details of such a Web Service, e.g. it must have a WSDL operation called GetRules responsible for returning the rules.

Document Store interface

The document store interface serves two purposes. It allows us to import the content of the document store as well as to export the modifications back to the document store. This interface is implemented by a Web Service of the host system.

Because of the size of the document store, the interface specifies that the Web Service must be capable of returning the relevant content of the document store in parts. Namely, the Web Service must realise two operations: (1) GetNumberOfNodes which returns the number of nodes in the document store and (2) GetNodesByID returning nodes within a given interval: e.g. nodes with IDs between 1–100.

Exporting the results is done by invoking the UpdateProperties operation of the Web Service. The input of this is a set of modifications that should be performed on the document store. UpdateProperties can be called multiple times, to support the option of performing the export operation in parts, whenever the reasoning process has too many results, i.e. too many new properties were inferred during reasoning.

Information Extractor interface

The Information Extractor is an autonomous component in a sense that it is expected to be developed independently from SREngine. If this component becomes more intelligent, i.e. a new data extraction algorithm is built in, the users of the SREngine system will automatically be able to use this when formulating rules.

To support this behaviour, we have created a registration mechanism which lets the Reasoner know about the external calls, i.e. calls that are executed not by the Reasoner, but the Information Extractor.

Registration information is described in the SRLang language and is contained in the configuration part of the rule input file. Here the user specifies which Web Service should the system invoke in order to execute a specific external call. Such a Web Service must implement two operations called ex_property and ex_calculate. The first is used for external calls where binaries of the documents are involved and thus their execution is considered to be expensive. The latter is used in other cases, e.g. for simple string manipulations. Examples for external calls are given in the upcoming sections.
The reply of the Information Extractor always contains the status field which can have three different values: success, failure and exception. In case of success the Information Extractor may provide some return values.

5.4 The SRLan Rule language

We now introduce the rule language of the SREngine system. First the basic syntactic components are described, then we explain the quantified operations of the language. Subsequently, we discuss the external calls and the first order logic semantics of the rules. The upcoming examples describe rules that classify documents based on some criteria: the profession of their authors, their content, etc.

Throughout this section we use the text format of the rules as opposed to the much more verbose XML format we do not detail in this chapter. The text format is actually based on the Prolog syntax [115]: the rules are Prolog expressions using appropriate operator declarations as connectives.

5.4.1 Basics of SRLan

The rule input file in Figure 5.2 consists of SRLan configuration elements and rules. Using configuration elements one can specify the active properties, i.e. new properties that have to be written back to the document store. An example can be seen below.

deducible(document_type).
deducible(has_author).

This describes that the results of the rules producing document_type and has_author properties will be exported to the document store. Other rules can be used during deduction, but their results are not exported directly. However, they can contribute indirectly to some exported results.

The other use of the configuration elements is to specify which Web Service to invoke in case of external calls. Below we can see two examples.

ex_calculate_location("contains", "http://152.66.71.114:1520/util").
ex_property_location("bulleted_list", "http://152.66.89.101/Tools").

Here we describe that in case of calls contains and bulleted_list the web services at the given locations should be invoked.

Now we turn our attention to the notion of SRLan rules. Similarly to Prolog, an SRLan rule consists of a head and a body separated by the <= character sequence. Below we show a simple example.

document_type(Document, "category:/documents/scientific") <==
has_author(Document, Author) and
has_profession(Author, "jobs:/education/teacher").  (5.1)

This rule states that a document should be classified as a scientific document if it has an author who is a teacher. Conditions has_author, has_profession and document_type correspond to properties in the Document Store. The latter is of special importance as it is used to describe the category of a document, i.e. rules with heads like this are classification rules.

Identifiers given as Prolog atoms, e.g. "jobs:/education/teacher", are called path expressions. These expressions identify nodes within the Document Store by navigating along the containment relation. In principle, the user could write node identifiers (numbers which uniquely identify nodes) here as well, but practically, path expressions are more readable. For compactness, if this does not cause confusion, we will use a simplified version of path expressions in this chapter: for example, we will simply write teacher instead of "jobs:/education/teacher".

In our next example we classify a document object as belonging to the contract category if it is a Microsoft Word document, its binary (i.e. the document itself) is a bulleted list and it contains the word “contract” somewhere in the document body:
This rule is a good example for illustrating that we can refer to information extracted from the content of the documents when creating rules. The truth values of the `ex_property` expressions above are given by the Information Extractor component (cf. Figure 5.2). As we have seen above, the technical details on how to invoke such an operation can be described in the configuration part of the Rule Store. The content of the input parameters of an external call varies from call to call, so the expert should know with what parameters she should invoke the given operation. We will discuss external calls in more detail in Section 5.4.3.

SRLang also supports the use of negation as failure, i.e. the closed world variant of the classical negation. In the example below we classify a document as a singleton if it has a name and it does not have a known sibling having binaries (note that this rule is „unsafe“ as negation is applied to non-ground literals; this can be avoided by using the `exists` quantifier, see below).

Note that we do not put any restriction on how many categories a document can be classified in. In this sense the SRLang language resembles more an ontology language than an object-oriented one.

### 5.4.2 Quantified operations

In SRLang we can express that a certain property holds for at least one element or for all the elements of a given solution set. The former corresponds to the existential (∃), the latter to the universal (∀) quantifier. For example, we can formulate a condition that every author of a document must have a specific property:

![Formula](5.2)

Here we say that if all the authors of a document are artists, then we classify it as an art document.

The `forall` construct can also be used to express more complicated conditions. For example, we can formulate a rule with a nested `forall` condition requiring that all degrees of all authors of a document are of certain kind:

![Formula](5.3)

Generally a `forall` expression has the following form, where variables \(X_1, \ldots, X_j\) must be present in expressions \(F_1, \ldots, F_n\):

![Formula](5.3)
This requires that we collect all solutions of $F_1$ and ... and $F_n$ in variables $X_1$, ..., $X_j$, and for each such solution we check that $T_1$ and ... and $T_k$ holds. Note that this implies that we use the closed world assumption with respect to the forall expression as opposed to the forall-like constructs of other related languages, such as the allValuesFrom in OWL [8].

We have already seen examples with implicit existential quantification. For example, rule (5.1) classifies a document to belong to a given type if at least one of the authors has the specified profession.

For the sake of symmetry we also allow explicit existential quantification, listing the quantified variables. Accordingly, rule (5.1) can also be given in the following way:

\[
\text{document_type}(\text{Document}, \text{scientific}) \iff \\
\text{exist Author::} \\
\quad \text{has_author}(\text{Document}, \text{Author}) \quad \text{and} \\
\quad \text{has_profession}(\text{Author}, \text{teacher}).
\] (5.4)

Using the exist construct also helps the SREngine system to warn the user in certain situations. Namely, if one of the variables in a condition is existentially quantified, but at runtime it is not instantiated, the SREngine gives a warning to the user (cf. floundering in Prolog [117]).

The general form of the exist construct is analogous to that of the forall expression:

\[
\text{exist } X_1, \ldots, X_j \text{ in } F_1 \text{ and } \ldots \text{ and } F_n :: T_1 \text{ and } \ldots \text{ and } T_k
\] (5.5)

### 5.4.3 External calls

SRLang gives us the possibility to access functions implemented in external libraries while evaluating the rules. The ex_property calls are used to execute different kinds of data extraction algorithms on one or more binaries. An ex_property has three parameters. The first is a string describing the operation we would like to invoke. The second parameter is a list which contains references to binaries and possibly to other objects we need for the evaluation. The third parameter is also a list. This list will contain the results after a successful execution. In the following we show some examples for the usage of ex_property:

\[
\begin{align*}
\text{ex_property("contains", ["(pr(5),treaty or contract)"],[])} \\
\text{ex_property("get_author",[Binary], [Author])} \\
\text{ex_property("basic_metadata", [Binary], [Date, Format, Language])}
\end{align*}
\]

In the first example we ask whether the given document contains the phrase treaty or contract in the paragraph 5. We can see that here we have two inputs and no output (as indicated by the empty list). Note that the input arguments can be fairly complex. Handling these is the task of the program code responsible for the execution of the operation in question.

In the second example we extract the author of the document and put it in the variable Author. In this example we have one input and one output.

In the third example we extract several pieces of data from the document (the creation date, the format and the language of the binary) and put them in the variables of the output list.

The ex_calculate calls have the same syntax as ex_property calls: the first argument specifies the operation, the second and third arguments describe the input and output parameters. The difference between the two kinds of external calls is that during the execution of an ex_calculate call binaries are not involved. This implies that while ex_property calls must be handled by an external component (as it has to efficiently access the binaries), ex_calculate calls can also be implemented within the SREngine itself. In fact, there are several predefined ex_calculate operations, including string manipulations, arithmetic, etc. Some examples are shown below.

\[
\begin{align*}
\text{ex_calculate("multiply", [A, B], [C])} \\
\text{ex_calculate("append", [FirstName, LastName], [Name])} \\
\text{ex_calculate("contains", ["bill", Name],[])}
\end{align*}
\]
In the first example we multiply numbers A and B and expect the result to be assigned to variable C. The second example concatenates two names, while in the third we examine whether the string Name contains bill as a substring.

To sum up, an external call can invoke an arbitrary Web Service which we consider a very flexible way to extend the capabilities of a rule based reasoning system.

5.4.4 Semantics

The evaluation of a set of SRLan rules with respect to a given document store results in a set of new document properties. This set is expected to contain exactly those properties that are entailed by the first order logic equivalent of the SRLan rules in question (using the closed-world assumption).

We now discuss how to transform SRLan constructs to first order logic formulae. SRLang rules which do not contain negation and forall constructs correspond to simple Horn clauses. The variables in the rules correspond to logic variables, the literals, numbers, and path expressions correspond to logic constants. The \( \iff \) connective denotes implication, while the and construct corresponds to conjunction \((\land)\). Accordingly, example rule (5.1) corresponds to the following logic formula:

\[
(\forall \text{Document, Author})(\text{document\_type}(\text{Document, scientific}) \iff \\
\quad \text{has\_author}(\text{Document, Author}) \land \\
\quad \text{has\_profession}(\text{Author, teacher}))
\]

The forall construct corresponds to a special kind of universal quantification. For example, the logic form of rule (5.2) is shown below.

\[
\forall \text{Document}(\text{document\_type}(\text{Document, art}) \iff \\
\quad \forall \text{Author}(\text{has\_author}(\text{Document, Author}) \implies \\
\quad \text{has\_profession}(\text{Author, artist}))
\]

The general form of the forall expression shown in (5.3) corresponds to the following logic formula:

\[
(\forall X_1 \ldots X_j)(T_1 \land \ldots \land T_k \iff F_1 \land \ldots \land F_n)
\]

The logical equivalent of the exist construct is the existential quantification. For example, rule (5.4) corresponds to the logic formula shown below.

\[
\forall \text{Document}(\text{document\_type}(\text{Document, scientific}) \iff \\
\quad \exists \text{Author}(\text{has\_author}(\text{Document, Author}) \land \\
\quad \text{has\_profession}(\text{Author, teacher}))
\]

The general form of the exist construct introduced in (5.5) is equivalent to the following logic formula:

\[
(\exists X_1 \ldots X_j)(F_1 \land \ldots \land F_n \land T_1 \land \ldots \land T_k)
\]

5.5 Executing SRLan rules

In this section we discuss the execution of the SRLan rules and the implementation details of the system. First we give a high-level overview of the execution process. Next, we discuss the properties of the bottom-up reasoner SREngine uses: we describe how we stratify the rules to ensure a sound reasoning process.

5.5.1 The SREngine execution process

The operation of the SREngine is performed in the following 6 steps:

1. process the initial configuration
2. fetch the content of the document store
3. fetch the rules
4. build layers
5. perform bottom-up reasoning
6. export the results

First, the system reads the configuration files where the general parameters are stored. These include important details on how to access the document store and from where to fetch the rules. In the next two steps we load these two sources, respectively. In step 4, we group the rules into layers (see more below). Next, we perform a bottom-up reasoning process and finally we export the results.

5.5.2 Bottom-up reasoning

In the SREngine system we have implemented a bottom-up reasoner. This choice, contrasting with the top-down execution mechanism of Prolog or any other resolution based logic programming system such as XSB [138], is justified by the fact that our task here is to produce all document properties that can be deduced from the document store and the rules (while in the top-down approach we look for the solutions of a single goal).

A further advantage of bottom-up reasoning is that it is much more robust than the top-down approach, e.g. in terms of ensuring termination (although termination problems may still occur [14]). As no serious attention needs to be paid to this aspect, rules can be formulated more freely and do not require Prolog expertise.

A disadvantage is that using bottom-up reasoning in the presence of negation as failure or the forall construct raises some problems. For example, even though at a given point it seems that a property holds for every author of a specific document (and thus the corresponding rule can be applied) this cannot be taken for granted. This is because a bottom-up reasoner can infer later that the document has another author who may not have the desired property.

These problems are actually avoided as we only allow set of rules that can be stratified [2] with respect to negation and forall. In this case the rules can be divided into layers in such a way that if a rule calls another through negation or forall, then this other rule has to be placed in a lower layer than the calling rule. If such a stratification can be found, then bottom-up reasoning can be safely applied to the layers one-by-one, starting from the lowest.

In the SREngine system we apply a slight generalisation of this technique. Namely, when building the layers, we consider every kind of dependency between the rules, not only the calls via negation and forall. This strategy is discussed below. Note that we realised that excluding non-stratified sets of rules is a strong restriction. However, we argue that (1) in the context of document management stratified rule sets seem to be sufficient and (2) this simplification makes the reasoning more scalable as opposed to other possible approaches such as using stable-model semantics for Logic Programs [42].

5.5.3 Layers

In the SREngine system, the layers are created in the following way. First we build the dependency graph $G_d$ from the actual set of rules. Each node in $G_d$ corresponds to a rule. An edge from node $A$ to node $B$ represents the fact that the execution of $B$ may depend on the execution of $A$, i.e. $A$ can possibly produce new facts which can trigger the rule corresponding to node $B$. As an example, the dependency graph of the rules shown in Figure 5.3 is presented in Figure 5.4. Here the three rules producing has_author properties are denoted by has_author1, has_author2 and has_author3 respectively.

Having constructed $G_d$, our next step is to partition this graph according to its strongly connected components. A strongly connected component (SCC) is a subgraph of $G_d$ such that, for every pair of vertices $A$ and $B$ there is a path from $A$ to $B$ and also a path from $B$ to $A$. In Figure 5.4 the strongly connected components are marked by dashed-lined boxes.

Based on the SCCs we build a reduced graph $G_r$. The vertices of this graph are the strongly connected components of $G_d$. There is an edge in $G_r$ from $A$ to $B$ if and only if there is an edge in $G_d$ from one of the vertices in the SCC corresponding to $A$ to one of the vertices in the SCC corresponding to $B$. 
As $G_r$ is acyclic by definition, a topological ordering of $G_r$ can be constructed, i.e. a full ordering on the nodes such that if there is an edge from $A$ to $B$, then $A$ precedes $B$ in the ordering.

The graph $G_r$ and a topological ordering of the nodes of $G_r$ is used for stratifying the initial set of rules. The set of rules corresponding to a node of $G_r$ will form a stratum. The ordering of the strata is determined by the topological ordering of $G_r$, so that the lowest value forms the bottommost layer, and so on. For example, a valid topological ordering of the graph in Figure 5.4 is indicated by the numbers in the figure: layer 1 is the bottommost, layer 4 is the topmost.

Given a stratification we execute the rules in the bottommost layer until they stop deducing new properties. We then repeat this process for all layers upwards.
5.6 Evaluation

In contrast with the traditional document classification approaches, in the SREngine system the classification rules are not necessarily used to classify documents because of similar content. For example, a category could be “Books of Hermann Hesse”, where an SRLang rule may classify a document to belong to this category just using the author meta-information. It is obvious that the books of Hermann Hesse do not necessarily have similar content from the traditional document classification point of view.

More closely related to our approach are the rule-based RDF languages mentioned in Section 5.2. These languages are expressive enough to capture most of the rules an expert would like to formulate in the context of document classification. However, we argue that the SREngine system provides a viable alternative to other systems because of the intuitive syntax, the rich set of modelling constructs (like \texttt{forall}), the bottom-up execution mechanism and the document extraction capabilities of the system.

We have carried out an initial performance evaluation with a relatively small number of rules, but huge amounts of meta-information in the document store. This is a typical setup as in real applications the content of the document store is several magnitudes larger than the number of rules. Our prototype implementation is built on top of the Sense/Net Portal Engine, an enterprise portal management system providing content management, application integration and collaboration facilities [82].

For our tests, the document stores were randomly generated. To each of these we applied a fixed set of rules. This sample rule set, containing 10 rules, was chosen to include the most important SRLang constructs, such as recursive rules, external functions, \texttt{forall}, \texttt{exist}, \texttt{not}, etc.

The tests were executed in the following hardware and software environment: Intel Pentium-M 1.7GHz, 512MB memory, Ubuntu Linux 7.10, SICStus Prolog 3.12.5.

The left hand side of Table 5.1 shows the properties of the document stores we used for testing. The columns show the size of the document store, the number of properties it contains and the average branching factor. This factor describes how many outgoing edges a node in the property graph contains on average.

<table>
<thead>
<tr>
<th>Size</th>
<th># of prop</th>
<th>Branch</th>
<th>New prop</th>
<th>Load time</th>
<th>Exec time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7KB</td>
<td>33</td>
<td>5</td>
<td>8</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td>3.3KB</td>
<td>81</td>
<td>5</td>
<td>16</td>
<td>0.02s</td>
<td>0.01s</td>
</tr>
<tr>
<td>19KB</td>
<td>541</td>
<td>6</td>
<td>80</td>
<td>0.06s</td>
<td>0.05s</td>
</tr>
<tr>
<td>348KB</td>
<td>10194</td>
<td>7</td>
<td>1315</td>
<td>0.75s</td>
<td>1.06s</td>
</tr>
<tr>
<td>697KB</td>
<td>20443</td>
<td>7</td>
<td>2641</td>
<td>1.40s</td>
<td>0.95s</td>
</tr>
<tr>
<td>1.2MB</td>
<td>35837</td>
<td>4</td>
<td>6506</td>
<td>2.50s</td>
<td>2.30s</td>
</tr>
<tr>
<td>2.4MB</td>
<td>69287</td>
<td>6</td>
<td>9889</td>
<td>3.60s</td>
<td>3.50s</td>
</tr>
<tr>
<td>3.9MB</td>
<td>112528</td>
<td>8</td>
<td>13299</td>
<td>5.10s</td>
<td>4.60s</td>
</tr>
<tr>
<td>12.6MB</td>
<td>359833</td>
<td>4</td>
<td>65491</td>
<td>13.60s</td>
<td>23.70s</td>
</tr>
<tr>
<td>25.6MB</td>
<td>720177</td>
<td>4</td>
<td>130683</td>
<td>25.50s</td>
<td>50.50s</td>
</tr>
</tbody>
</table>

Table 5.1: Results of the performance analysis

The right hand side of Table 5.1 shows the results of the test. Here, the first column indicates the number of new properties deduced by the SREngine during the execution. The second column contains the time it took to read the content of the document store. Finally, the last column contains the execution time, i.e. the time needed for the deduction.

The results clearly indicate that for a relatively small set of rules the SREngine system has a fairly good performance even for big document stores.
5.7 Future Work

Up to now, the SREngine system supports only “positive” consequences, i.e. the result of the reasoning process is the list of new edges that should be added to the document store. As an extension, we plan to provide the user the ability to create rules that lead to “negative” consequences, i.e. they can delete metadata from the document store by specifying appropriate rules.

So far we have considered all the properties multi-valued, i.e. a given document object can have several edges attached to it with the same property. However, some properties are typically single valued, such as the has_binary. Handling these and the related consistency problems is future work.

Finally, we need to find practical applications where the SREngine system is used in real corporate environments providing us relevant feedback about the system usability, performance and further extension possibilities.

5.8 Conclusion

In this chapter we presented the SREngine system, which is a generic framework that can be used to infer new document properties in a document store, using a set of rules.

We have presented the main components of the SREngine: the Knowledge Base which stores the rules and the relevant parts of the document store, the Reasoner, which performs a bottom-up reasoning process to infer new properties and the Information Extractor, which provides data extraction functionality, to be used when formulating rules. Here, we have also introduced the interfaces the SREngine system uses for communicating with the host system managing the document store.

We described the SRLang language, which is a Prolog based language for describing user friendly rules involving documents. We have introduced the main constructs, the quantified operations and the external calls. We have also presented the semantics of SRLang rules.

Next, we have discussed the bottom-up reasoning process we use to execute SRLan rules. Finally, we have evaluated the implementation of the SREngine system by presenting some basic performance results.

As a conclusion we believe that the SREngine system provides a robust, extensible framework for deducing new meta-information about documents. The SREngine system does not require Prolog expertise and can be used together with various host systems.
Summary

This thesis described work on the applicability of logic programming in various knowledge intensive applications. To conclude we summarise the contributions of the author, together with a brief evaluation of the significance of the results.

In the first part of the thesis I showed how Logic Programming can be used to support the two modelling methods of Description Logic: the open world and the closed world approach. I designed and implemented a transformation framework in the DLog system the aim of which is to compile Description Logic axioms into Prolog programs (Chapter 2). I also carried out detailed performance analysis. From the results we can conclude that DLog is a viable alternative to the existing state-of-the-art open-world Description Logic reasoners as it provides better performance and scalability. Below we summarise the results of this work.

**Thesis 1.** Based on [99] and [142], I developed the theoretical basis of the DLog description logic reasoner and proved the soundness and completeness of the proposed algorithms. I was the lead architect and developer of the DLog system. [95, 104, 103, 89, 102, 88, 85]

- **Thesis 1.A.** I specialised the PTTP technology for DL-clauses.
- **Thesis 1.B.** I defined a generic compilation scheme for DL-programs.
- **Thesis 1.C.** I developed a series of optimisations for \( \text{SHIQ} \) knowledge bases that result in a much more efficient Prolog translation compared to the generic compilation scheme.

The prototype of the DLog description logic reasoner is available for download from the following web address: [http://dlog-reasoner.sourceforge.net](http://dlog-reasoner.sourceforge.net). We expect the DLog system to be used in applications where description logic reasoning is needed on large amounts of data. Furthermore, we believe that some methods and algorithms described in the Thesis can also be used in other description logic reasoners to make them more efficient.

My other contribution in the context of Description Logic is a hybrid modelling methodology that uses Description Logic models on top of Object-oriented models together with the corresponding closed-world execution mechanism (Chapter 3). The results are summarised below.

**Thesis 2.** For the SINTAGMA information integration system I developed a modelling methodology and the corresponding execution mechanism which combines the advantageous features of the DL and the object-oriented modelling paradigms. [136, 91, 12, 86, 92, 87]

- **Thesis 2.A.** I defined a modelling methodology which describes how to use Description Logic and UML models during information integration in a combined fashion.
- **Thesis 2.B.** I defined a compilation scheme that translates the description logic constructs of SILan, the SINTAGMA modelling language, into query goals.

The SINTAGMA system has been put to experimental use at several institutions, including the National Széchényi Library and the Hungarian News Agency.

In the second part of the dissertation I presented two applications in which Logic Programming plays a crucial role.
I designed and implemented the Match system, a generic plagiarism detection framework that can be used to determine the similarity degree of source programs (Chapter 4). The results of this work are summarised below.

**Thesis 3.** I developed the theoretical basis of the Match plagiarism detection system. Match uses the structural properties of the source codes to determine their similarity degree. I designed and implemented the system. [93, 90]

- **Thesis 3.A.** I designed the comparison framework of the Match system.
- **Thesis 3.B.** I defined the similarity degree of two source programs.
- **Thesis 3.C.** I created algorithms and the related heuristics for determining the similarity degree of directed acyclic graphs.

The Match application has been used at the Budapest University of Technology and Economics to successfully detect plagiarism in homework assignments for the past eight years. In addition to the Prolog language interface I implemented, Match has been extended to work with SML source programs as well. This makes it possible to use the system for both languages we teach within the Declarative Programming course. Based on experience we can say that Match significantly reduces the manual work needed for detecting plagiarism: in several cases it has drawn attention to pairs of students who, by their own admission, spent several hours on trying to conceal the fact of plagiarism.

Finally, I designed and implemented the SREngine system, which provides a generic framework that can be used to infer new document properties in a document store, using a set of rules (Chapter 5). The related theses are summarised below.

**Thesis 4.** I developed the theoretical basis of the SREngine rule based reasoner which automatises the annotation of resources. I designed and implemented the system. [96, 94]

- **Thesis 4.A.** I developed the SRLang logic based language which can be used to describe rules concerning meta-information.
- **Thesis 4.B.** I designed and implemented the RuleEngine system.

The RuleEngine system has been put to use with the Sense/Net Portal Engine as its host system, which is an enterprise portal management framework. This demonstrated the applicability and advantages of the chosen approach. However, we need to do further tests and improvements to make RuleEngine applicable in real life circumstances.

The link between the above presented results is that they all introduce novel semantic technologies, they apply logic programming techniques in general and the Prolog language in particular, and they have been actually implemented in the form of various applications. Thus this PhD Thesis demonstrates that Logic Programming can be successfully used for developing novel algorithms and tools for semantic technologies in various important application areas.
Bibliography


