FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Development of a Belief Merging Framework for dlvhex

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Christoph Redl, BSc.
Matrikelnummer 0525250

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: O.Univ.Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Mitwirkung: Dipl.-Ing. Thomas Krennwallner

Wien, 1. Juli 2010

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Christoph Redl
Kieslingstraße 9
3500 Krems

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juli 2010

(Unterschrift Verfasser)

Abstract

In many fields it is common to work with several data sources. This enables us to enlarge the total size of the knowledge base, leading to an information gain. If the sources are overlapping but not equivalent, this will allow the user to combine the advantages and use the most useful parts of each of the bases.

Belief revision deals with the incorporation of new information into an existing belief base. During this process, the new entry has absolute priority, which means that it will be *true* in the revised base for sure. In contrast to that, *belief merging* deals with the aggregation of several sources. Unfortunately, naive union can lead to an inconsistent set of beliefs. Therefore the goal is to find a new base that is consistent and as similar to the existing ones as possible, according to some distance function.

A practical application is the merging of medical data about a patient that were collected by different health institutes. But knowledge fusion is not only useful in medical settings. A very common task is *ontology integration* in the Semantic Web or the incorporation of employee or customer databases in case of company fusions. Other examples include the union of multi-dimensional index structures and the combination of classifiers that were trained upon disjoint sets of samples.

In this thesis our case study will be *judgment aggregation*, which is a topic from the field of social choice theory, where the judgments of several individuals need to merged into a group decision.

The open-source software dlvhex is a reasoner for an extension of logic programs under the answer-set semantics, called HEX programs. Basically it provides two extensions, namely higher-order atoms and external atoms. Higher-order atoms enable the programmer to use variables instead of symbols as predicates. External atoms allow a bidirectional communication between logic programs and external sources of computation like relational databases, XML files and RDF ontologies. For this purpose, dlvhex can be extended with user-defined *plugins* provided as shared object libraries.

In this thesis, the possibility of writing plugins will be exploited to implement a framework for belief merging tasks. The input is a set of HEX programs that deliver sets of answer sets which represent the belief sources to be merged. Internally these programs can evaluate external atoms to access arbitrary data sources. In the merging task description the user defines how to combine these sources. For this purpose he or she can use *merging operators* that are applied on

belief bases or recursively on the result of previous operator applications. This leads to a tree-like structure, where the leaf nodes are belief bases and the inner nodes are merging operators. This is very similar to the way we evaluate arithmetic expressions. The final result of a merging task is the result of the topmost operator.

In the literature there were already defined operators for belief revision and update tasks. This idea can easily be generalized, such that not only a single formula is incorporated into an existing knowledge base, but complete belief bases are fused. However, to keep the framework flexible the user can not only select among a fixed set of operators, but can also implement custom ones that can be dynamically added to the framework. This makes the framework useful for any kind of merging task as long as suitable operators are provided.

The actual benefit of the framework is that it takes the burden of repeating routine tasks from the user. The information flow through the different merging operators is managed automatically, so that the user can concentrate on the most interesting and relevant part, namely the development and optimization of the merging procedures. Further the framework allows to restructure the merging plan (i.e., the selection of operators and order of their applications) in a user-friendly merging language. This way it is possible to experiment with several settings and compare the results without manually reimplementing the flow of information after each modification.

Zusammenfassung

Heutzutage arbeitet man in vielen Anwendungsbereichen mit mehreren Datenquellen, deren Inhalte im Zuge der Abfragen kombiniert werden. Diese Vorgehensweise ermöglicht es dem User die Informationen und Vorteile der einzelnen Provider miteinander zu verbinden. Sind die Inhalte teilweise überlappend, aber nicht vollkommen deckungsgleich, so erhöht dies den Gesamtwissensstand.

Das Gebiet *Belief Revision* beschäftigt sich mit der Aufgabe, neue Information in *eine* vorhandene Wissensbasis konsistenzerhaltend einzubringen. Dabei hat das neue Wissen absolute Priorität, das heißt es wird nach dem Revisionsprozess mit Sicherheit in der Datenbasis enthalten sein. Im Gegensatz dazu haben wir bei *Belief Merging* mehrere Quellen, die miteinander zu kombinieren sind, ohne dass dabei bestimmte Informationen eine höhere Wichtigkeit haben als andere. Die naive Vereinigung kann auch dabei Inkonsistenzen herbeiführen, weshalb intelligentere Merging-Strategien notwendig sind, die einerseits die Widerspruchsfreiheit garantieren, andererseits aber versuchen, die Distanz zwischen Ergebnis und Quellen zu minimieren.

Eine mögliche Anwendung ist das Vereinigen von medizinischen Patientendaten, die von verschiedenen Gesundheitsinstitutionen erhoben wurden. Eine weitere kommt aus dem Bereich des Semantic Web, wo Ontologien sehr beliebt sind und oft verwendet werden. Dabei stellt sich die Aufgabe, mehrere teilweise überlappende Ontologien in eine einzelne zu vereinen. Ein aus der Wirtschaft kommendes Szenario ist die Zusammenführung von Datenbanken im Falle von Firmenübernahmen.

In dieser Arbeit wird jedoch ein anderes Fallbeispiel näher beleuchtet, nämlich *judgment aggregation*. Dieses Thema stammt aus der *social choice theory* und befasst sich mit der Vereinigung von verschiedenen Einzelmeinungen um schließlich eine Gruppenentscheidung treffen zu können.

Die Open-Source-Software dlvhex ist ein Reasoner für eine Erweiterung von logischen Programmen unter der Answer-Set-Semantik, die als HEX-Programme bezeichnet werden. Prinzipiell erweitern diese die herkömmliche Semantik auf zwei Arten. Einerseits werden Atome höherer Stufe unterstützt, die es dem Anwender erlauben, Variablen statt Prädikatensymbole zu verwenden. Andererseits wurde der Formalismus um externe Atome erweitert. Diese ermöglichen einen bidirektionalen Datenverkehr zwischen dem logischen Programm und externen Quellen wie etwa relationalen Datenbanken, XML-Dateien oder RDF-Ontologien. Zu diesem Zweck kann dlvhex

mit benutzerdefinierten Plugins erweitert werden, die in Form von Shared-Object-Bibliotheken installiert werden.

Ziel dieser Arbeit ist es, ein Framework für dlvhex zu entwickeln, das den Benutzer bei verschiedensten Merging-Szenarien unterstützt. Dazu wird die Möglichkeit zum Schreiben von Plugins ausgenutzt. Eingabe für das Framework ist eine Menge von HEX-Programmen, deren Answer-Sets als das im Programm gespeicherte Wissen betrachtet werden. Intern können diese Programme beliebige externe Datenquellen kontaktieren. Weiters definiert der Benutzer wie die einzelnen Quellen verknüpft werden sollen. Dazu steht ihm eine Menge von vorgefertigten Merging-Operatoren zu Verfügung, die auch erweitert werden kann, falls sich kein passender Operator darin findet. Diese Operatoren werden schließlich, ähnlich wie im Falle von arithmetischen Ausdrücken in der Mathematik, in einem baumartigen *Merging-Plan* angeordnet, wobei die Blattknoten die Datenquellen und die inneren Knoten die Operatoren darstellen. Das Ergebnis des Merging-Tasks wird vom äußersten Operator zurückgegeben.

In der Literatur wurden bereits einige Operatoren für Belief Revision und Update entwickelt. Die Idee dieser Operatoren kann oft sehr einfach für Belief Merging-Szenarien verallgemeinert werden. Einige dieser Operatoren wurden im Framework bereits vorgefertigt implementiert. Dennoch versteht sich diese Auswahl nur als Beispiel. Um für maximale Flexibilität zu sorgen, steht es dem Benutzer frei, vorhandene Operatoren zu verfeinern oder neue zu implementieren.

Der Hauptnutzen des vorgestellten Frameworks ist die Entlastung von Routineaufgaben. Der Informationsfluss von den Quellen zu den Operatoren und zwischen einzelnen Operatoren wird völlig automatisiert verwaltet. Dadurch kann sich der Benutzer auf die wesentlichste Aufgabe konzentrieren, nämlich die Merging-Strategie im engeren Sinne zu optimieren. Weiters erlaubt es das Framework, nachträgliche Änderungen des Merging-Plans auf eine sehr benutzerfreundliche Art und Weise. Damit muss die Zusammenführung mehrerer Datenquellen nicht wiederholt werden, wie das bei einem manuellen Ansatz der Fall wäre. Es ist damit möglich, viele verschiedene Strategien auszuprobieren und Ergebnisse zu vergleichen.

Contents

List of Tables

List of Figures

Danksagungen

An dieser Stelle möchte ich mich bei jenen Menschen bedanken, die zum raschen Abschließen dieser Arbeit beigetragen haben.

Mein Dank gilt vor allem meinem Betreuer Prof. Thomas Eiter, der sich in zahlreichen Meetings die Zeit nahm, mir beratend zur Seite zu stehen, wertvolle Verbesserungsvorschläge zu geben und meine Fragen zu beantworten. Ebenfalls mitgewirkt hat Thomas Krennwallner, der bei implementierungstechnischen Fragen stets kompetente Auskunft erteilen konnte.

Dem FWF (Fonds zur Förderung der wissenschaftlichen Forschung[1]) danke ich für die freundliche finanzielle Unterstützung des Projektes *Modular HEX-Programs* (P20841), sowie dem WWTF (Wiener Wissenschafts-, Forschungs- und Technologiefonds[2]) für die Förderung des Projektes *Inconsistency Management for Knowledge Integration Systems* (ICT 08-020), in deren Umfeld ich diese Arbeit schreiben durfte. Dadurch wurde es mir erst ermöglicht an diesem interessanten Thema zu arbeiten.

Weiters haben auch alle weiteren Mitglieder der Arbeitsgruppe für wissensbasierte Systeme ihren Teil dazu beigetragen, mein Interesse auf Logik und logikorientierte Programmierung zu lenken. Dies gilt vor allem für Uwe Egly, Hans Tompits und Michael Fink, deren spannende und kurzweilige Lehrveranstaltungen in den letzten Jahren für mich der Anlass waren, meine Abschlussarbeit letztendlich in diesem Arbeitsbereich zu schreiben.

Einen Beitrag haben auch meine Studienkollegen geleistet. Jeder auf seine Art und Weise. Sei es, indem sie mich durch ihren Ehrgeiz angespornt haben die Ziele hoch zu stecken und Schritt halten zu wollen, oder weil sie in vielen belanglosen Gesprächen für Spaß gesorgt und so das Studentenleben lockerer gemacht haben.

Nicht zuletzt bedanke ich mich natürlich auch bei meiner Familie, die es ertragen hat dass ich in den letzten Jahren trotz körperlicher Anwesenheit oft mit meinen eigenen Gedanken beschäftigt war. Vor allem gilt dieser Dank meinen lieben Eltern, Karl und Anita Redl, die mich während meiner Studienzeit sowohl finanziell wie auch moralisch unterstützt haben und ohne die der Weg weitaus beschwerlicher gewesen wäre.

[1] `http://www.fwf.ac.at`
[2] `http://www.wwtf.at`

Imagine if every Thursday your shoes exploded if you tied them the usual way. This happens to us all the time with computers, and nobody thinks of complaining.

Jef Raskin

Chapter 1

Introduction

1.1 Motivation

Knowledge-based applications often use information from several sources since in practice, it is rarely the case that there is one single point of truth. Because of different groups working on similar projects, competing commercial providers who share the market, different opinions and understandings of a certain topic, errors of measurement or simply human errors, there often exist similar but not equivalent knowledge sources.

As a first motivating example consider ontologies formalized in description logics. Ontologies are descriptions of the hierarchical relations between the concepts of a certain domain. In biomedicine, which is one of the most important but not the only application domain, there exist ontologies for viruses and other pathogenic agents. This is useful in case of a pandemia where it is necessary to know the relations of the rapidly changing virus mutants in order to know the original type where combating can begin.

A further medical example for an ontology is the representation of anatomical structures. It is obvious that such data is hierarchically organized. An organism consists of several organ system, where each of them consists of several organs, which in turn have several tissues. This relationship could be further extended into histological structures and enriched with lots of details. But the basic idea should be clear at this point. Other prominent medical ontologies include the GeneOntology (GO)[1] and SNOMED[2]. Also in non-medical applications ontologies are widely used, for instance in Semantic Web applications. Many of them were developed without a certain application in mind but act as a *formal knowledge repository* for future use cases [Horrocks, 2008].

Non-medical scenarios with multiple data sources include economic applications. An example is the fusion of companies where it becomes necessary to incorporate their information systems and especially their databases into one.

However, we will use another example for our case study in Chapter 6. Judgment aggregation deals with the merging of judgments by several individuals into a group decision. Practical scenarios of this topic, that comes from the field of social choice theory, can be found in court

[1] `http://www.geneontology.org/`
[2] `http://www.ihtsdo.org/snomed-ct/`

cases, planning of group activities, like holiday trips or meetings, and the selection of best qualified candidates by juries.

We will further take a closer look at a special case of judgment aggregation, namely the unification of diagnoses from several experts in propositional abduction problems. This is demonstrated using the circuit diagram of a logical hardware component.

1.2 Task Description

If multiple ontologies for the same domain exist, one can imagine that they are usually similar to each other, but not entirely the same. In such cases it would be easy to trust one of them completely and forget the others. However, we may unnecessarily lose a huge chunk of information. A much better approach is to incorporate the sources such that a single point of truth is built, where we use as much of the information as possible. In other words, we want to take the best parts of each source and combine the advantages.

A formal definition for this rather informal requirement is given in this thesis. It should be clear at this point that this task is far from trivial. Difficulties arise from a variety of reasons. First, it can not be assumed in general, that the data representation formalisms used in the information sources are compatible to each other. As a trivial example, imagine two databases storing a currency value somewhere. If several sources store the values in different currencies, a conversion must be done before the information can be merged.

More complex types of incompatibility can occur if the databases store essentially the same information, but were built upon different data models. Since there is not *one* correct modeling, we can not say that we trust one source more than the others but the databases must be made compatible in order to merge their content.

An even greater problem arises when different sources use entirely different representation formalisms. For instance, one stores its contents in a relational database and another one in an XML file.

Unfortunately, multiple sources are not always consistent even if they are superficially compatible. Imagine two relational databases, each storing customer data. Further assume that they use exactly the same data model. Then intuitively one could believe that it is straightforward to incorporate them. But if we simply perform a kind of union operation in order to combine the single data sources, logical inconsistencies can arise. In case of human errors for instance, several values could emerge where there should be only one. This can lead to a violated key constraint.

1.3 Approach

The gist of the previous section is the following. Knowledge sources first need to be made compatible in the sense that they need to share the same vocabulary and information representation formalism. This means that constants need to be adjusted where this is necessary, or tables need to be restructured in order to make data models equivalent.

The second subtask is to make the actual *data entries* consistent. If contradictions between information sources are discovered, parts of the belief bases need to be excluded. From the user's point of view, it is desirable to keep as much of the information as possible without introducing contradictions. This requirement reminds of several strategies that implement a kind of *maximal consistent subset (MCS)* operator. One of them is *Winslett's approach* which was introduced in [Winslett, 1988].

However, keep in mind that several other operators like WIDTIO's or Ginsberg's operator were also studied in the literature. The choice of a suitable one is strongly application dependent, i.e., there does not exist *the* best merging operator.

Therefore, the first goal in this thesis is the development of a theoretical framework for generic belief merging tasks. This framework shall be concrete enough to provide a solid basis for applications to build upon. At the same time it needs to be abstract enough to provide flexibility. This is necessary since the framework is not developed for a certain merging task, but must be applicable in many different scenarios. For this purpose, we will design it in a way that allows the user to extend it with custom merging procedures very easily. To summarize, the framework manages routine tasks that are similar in all merging applications, e.g., information flow management. The merging algorithms themselves, which are application dependent, are not part of the framework, but can be defined according to the specific requirements.

The next step is the development of a language for defining what we will call *merging plans*. This is a declarative representation of a certain merging scenario. Within the merging plan, the application developer can make use of the capabilities of the framework in order to specify his or her task. For this purpose, predefined or customized *merging operators* can be arranged hierarchically, which results in a tree-like structure similar to syntax trees representing arithmetic expressions. The final result of a merging plan is the return value of the topmost operator. Since the framework supports operators of arbitrary arity n, it especially also provides an elegant mechanism for filtering and similar tasks, realized by using unary operators.

In the practical part of the thesis, we will implement the developed framework in form of a plugin for the reasoner dlvhex[3]. dlvhex is a solver for HEX programs, which implement an extension of the answer-set semantics. Compared with pure answer-set reasoners like DLV[4], the HEX semantics adds support for higher-order atoms and external atoms. The latter are predicates with a semantics that is defined in an external source of computation rather than within the logic program itself.

The plugin will support a special user-friendly syntax for defining merging plans, as defined as part of the theoretical framework. They consist of a common vocabulary for all belief sources, a mapping for each input belief base that make them syntactically compatible, and a hierarchical arrangement of merging operators. When a task description is executed by the framework implementation, the merged belief set will be computed automatically. We will show that this corresponds to the return value of the topmost operator, as we have defined it in the theoretical part.

The actual intention of the developed framework is to give the user the possibility to try out several merging strategies without dealing with technical details like coordinating the information flow. This is done by hiding those details behind the merging plan language. This allows him to focus on the most interesting part, namely designing and evaluating the merging operators themselves.

Finally, the implementation of the framework will be used to implement Dalal-style operators [Dalal, 1988]. This will be illustrated with an application scenario from social choice theory, namely judgment aggregation [Eckert and Pigozzi, 2005]. That is, several experts have possibly different meanings about some question or topic. The task is to make a consistent group decision that respects some application dependent side constraints. We will see that this task can be elegantly solved using the framework.

1.4 Advantages

The framework developed in this thesis is one of the few practically implemented systems. Due to its generic design, it is the most general one by June 2010, as far as the author knows. Since

[3] `http://www.kr.tuwien.ac.at/research/systems/dlvhex`
[4] `http://www.dlvsystem.com`

it was not designed for a concrete merging task, the developed plugin can be used for a large variety of application scenarios. Examples include judgment aggregation, merging of decision diagrams (e.g., in medicine), and merging of multidimensional index structures. Even though the framework comes with a few predefined merging strategies, the user can easily extend it with custom ones.

The possibility to specify merging tasks declaratively makes it very easy to experiment with different algorithms and parameters and to evaluate the results empirically. If the best settings have been found using the framewok, a hard-coded reimplementation due to performance reasons can be considered.

For the implementation of the framework we will develop the concept of so called *nested* HEX *programs*. Informally speaking, it allows us to call subprograms while some HEX program is evaluated. Then their results are computed *independently* from the host program and can be accessed from the caller afterwards. This enables us to reason not only within single answer sets, but also on the level of sets of answer sets. Even though this is rather a technical detail and the user will not directly use it when working with the merging tool, it turns out that this feature is very useful independently from the merging framework. For instance, it allows us to answer queries using cautious or brave reasoning, even if the reasoner does not directly support this, or majority-based reasoning.

1.5 Discrimination from Belief Revision

The related field of belief revision deals with the incorporation of a new chunk of information into *one* existing knowledge base. During this process the new information has absolute priority, i.e., the new formula will follow for sure from the revised theory. Existing beliefs are given up if this is necessary to stay consistent.

In contrast to that, in belief *merging*, we have in general no formula with absolute priority (even though one could define an operator with similar semantics). Normally we just have *several* sources, and the task is to make a good compromise. This means we take a larger or smaller part of each input bases to set up the new knowledge base. A more formal definition of belief revision and merging will be given in Section 4.2.

1.6 Structure of this Thesis

The remaining chapters are organized as follows. Chapters 2, 3 and 4 are held very theoretical whereas 5 and 6 form the practical part. The following Chapter 2 summarizes the basic concepts of traditional logic programming, the stable model semantics, the answer-set semantics and finally the HEX semantics. It further introduces the notation that will be used in later chapters. The history of logic programming is shortly presented in order to demonstrate how the formalisms were generalized and extended over time. This makes it easier to see how they can be further extended in Chapter 3. There we introduce so called *nested* HEX *programs*. This is the execution of HEX programs within others and the access of their answer sets from the containing program, also called the *host program*.

Chapter 4 introduces the theoretical foundations of the belief merging framework. Notations that will be used later on as well as the merging task will be described in detail on an abstract level.

We will need the mechanism of nested HEX programs in Chapter 5, where the concepts of Chapter 4 are concertized and their implementation as dlvhex plugin is depicted. We will not go into technical details since the plugin is open-source and documented anyway, but the principles

of the implementation will be discussed and the translation mechanism of merging plans into semantically equivalent HEX program is explained in detail.

The practical part if completed by a description of possible application scenarios in Chapter 6. We first develop a merging operator that is very famous in literature, called *Dalal's operator* [Gabbay et al., 2009]. Then we use this operator to solve the problem of judgment aggregation. Finally we give a short overview about other possible application scenarios.

The thesis concludes with Chapter 7, where the main results are recapitulated and possible extensions and starting points for future work are shown.

Of course the falsity of the fact
that you believe it is red implies
that you don't believe it is red.
But this does not mean that you
believe it is not red!
Raymond Smullyan

Chapter 2

Preliminaries

Before we start with the actual topic of this theses we need to recapitulate the preliminaries. This chapter is intended to give an introduction to logic programming in general and HEX programs under an extended answer-set semantics in particular. Further it will present the notation we are going to use.

First we will describe the paradigm of *declarative programming*. Then we continue with one variant, namely logic programming. We shortly describe the concept of classical logic programming to show the historical roots and then bridge to modern approaches like programming under the *answer-set* semantics.

Finally we introduce HEX programs and point out the features that will form the basis for the framework that is developed in the later chapters.

2.1 Classical Logic Programming

Logic programming has a long history and thus several different semantics have been developed. The ultimate goal was to use logic in order to define *what* to compute rather than *how* this is actually done. This is called *declarative programming* in contrast to classical *procedural programming*. Whereas the latter paradigm is built upon machine principles, declarative programming is more problem centered. Some of the formalisms are somehow mixtures of declarative and procedural concepts, especially the early approaches, while newer developments are purely declarative. Today, logic programming is established as an important programming paradigm. Especially in the field of artificial intelligence, typical search problems are often solved by implementing appropriate rule-based programs. Even optimization problems like *graph colorability* can be elegantly be solved by logic programming.

When we think of logic as foundation for programming languages, we first have to decide *which* logic or *what* part of it we want to use. Several kinds of logic programming have been introduced, each with its own properties and restrictions. Everything started with what is nowadays called *classical logic programming*. We will introduce this type first since modern approaches are built upon.

In the following sections and chapters we presuppose familiarity with first-order logic and use the following notation.

Definition 2.1. A first-order signature $\Sigma = \langle \Sigma_v, \Sigma_p, \Sigma_c \rangle$ consists of a set of variables Σ_v, a set of predicate symbols Σ_p and a set of constant symbols Σ_c from a first-order vocabulary ϕ.

Resolution

An important concept that was necessary for actually using logic as programming paradigm is that of *resolution*. It is introduced in [Robinson, 1965] and unites the principles of *substitution* (i.e., replacement of variables by terms) and *truth-functional analysis* into one step, namely checking if two terms can be *unified*, i.e., if they can be made equal. This is the basis for automatic deduction and proving first-order sentences. Robinson presents a simple iterative resolution algorithm in his paper, which is basically suitable for being implemented in an interpreter for logic programs. However, in the meantime more advanced unification algorithms were developed. We follow the notation of [Martelli and Montanari, 1982] for formal definition of substitutions and the resolution problem.

Definition 2.2. Let $\mathcal{T}$ be the set of terms and $\mathcal{F}$ the set of formulas over a first-order signature Σ. A substitution σ is a set of tuples over $\Sigma_v \times \mathcal{T}$, denoted as:

$$\sigma = \{(X_i, t_i), \ldots, (X_n, t_n)\}$$

The result of the application of σ on a formula $F \in \mathcal{F}$, denoted as $F\sigma$, is a formula F', where each free occurrence of a variable X_i has been replaced by the according term t_i.

The algorithmic details can be ignored, but the principle is demonstrated by Example 2.1.

Example 2.1. Let $F = f(a, X) \wedge g(h(X), Y)$ and $\sigma = \{(X, b), (Y, c)\}$.
Then $F\sigma = f(a, b) \wedge g(h(b), c)$.

Now we go a step further and introduce the resolution problem. Informally speaking, that is the question if two terms, possibly containing variables, can be made equal by a suitable substitution [Baader and Snyder, 1999].

Definition 2.3. Let t_1 and t_2 be two terms. A solution to the resolution problem is a substitution σ s.t. $t_1\sigma = t_2\sigma$. σ is called a *unifier* of t_1 and t_2.

This is again demonstrated by an example (2.2).

Example 2.2. Let $t_1 = f(a, X)$ and $t_2 = f(Y, Z)$.
Then one possible unifier is $\sigma = \{(X, a), (Y, a), (Z, a)\}$.

Definition 2.4. Let σ, σ' be unifiers of terms t_1 and t_2.
Then σ is *more general* than σ' iff $\exists \sigma''$ s.t. $\sigma' = \sigma\sigma''$.

When solving a resolution task, one usually looks for a *most general unifier* (*mgu*).

Definition 2.5. A unifier σ of t_1 and t_2 is called *most general unifier* iff σ is a unifier of t_1 and t_2, and σ is more general than each unifier σ' of t_1 and t_2.

At this point we have laid down the concept of resolution for first-order formulas. But it needs a further development in order to use this mechanism as a programming paradigm. This is demonstrated in the next subsection.

Resolution as Programming Formalism

Taking all these achievments together, Kowalski observed how resolution can be used as programming formalism [Kowalski, 1974]. He uses a restricted version of first-order logic to formalize *rules* of the following form:

$$r = H_1, \ldots, H_m \leftarrow B_1, \ldots, B_n.$$

The syntax was already adjusted to today's Prolog syntax.[1] H_i are the head atoms and B_i are the body atoms. We will also write $H(r)$ and $B(r)$ to denote the set of head resp. the set of body atoms in rule r. Note that the term *atom* forbids the usage of any kind of negation. This is one of the characteristics of classical logic program that distincts it from later formalisms.

A logic program is simply a set of rules, where the rules can be partitioned into several classes depending on n and m (see Table 2.1).

n	m	Type
0	≥ 1	fact
≥ 1	0	goal
0	0	halt statement ($\square$)
arbitrary	≤ 1	Horn
arbitrary	1	definite Horn

Table 2.1: Rule types

For the remaining part of this section we will focus on Horn rules. In his original paper, Kowalski restricts programs to those containing only Horn rules since Robert Hill has shown that they are sufficient for defining all computable relations over a given universe. Thus we can define:

Definition 2.6. A classical logic program P is a set of Horn rules.

To compute the result of a program P, Kowalski uses resolution as introduced above and suggests the following procedure. Start with the goal statement

$$\leftarrow G_1, \ldots, G_n.$$

Then select a rule

$$H \leftarrow B_1, \ldots, B_{n'}.$$

such that there exists a substitution σ with $H\sigma = G_i\sigma$ for an $1 \leq i \leq n$ (i.e., H matches one of the G_i). Then the goal statement is replaced by

$$\leftarrow (G_1, \ldots, G_{i-1}, B_1, \ldots, B_{n'}, G_{i+1}, \ldots, G_n)\sigma.$$

This procedure is repeated until $\square$ (the halt statement) can eventually be derived or all deduction trials failed. Note that backtracking is necessary in general.

The question to answer is whether there exists a computation such that the halt statement can be derived. If this is the case, the computation is said to be successful, otherwise it fails. Using this machinery one can write goals in order to check if certain atoms follow from the program.

[1] The symbol $\leftarrow$ can be read as implication from right to left. Since it is not in the ASCII character set, actual implementations use strings like :-. However, we prefer the mathematical notation in this thesis.

Example 2.3. Consider the following program

$$\begin{aligned} P = \{ & fatherOf(jack, joe). \\ & fatherOf(joe, sam). \\ & grandfatherOf(X, Y) \leftarrow fatherOf(X, Z), fatherOf(Z, Y).\} \end{aligned}$$

Then the query

$$\leftarrow grandfatherOf(jack, sam).$$

is answered with *true* since it can first be reduced to

$$\leftarrow fatherOf(jack, joe), fatherOf(joe, sam).$$

and next each of the remaining atoms can be reduced to the halting statement using one of the two facts.

This is essentially what Prolog implemented in the early beginnings of logic programming. It concludes the formal definition of classical logic programming on the basis of resolution.

Least Fixed Point Semantics

In the last section we described Kowalski's resolution method for deriving atoms from logic programs. However, this method has several drawbacks. The most obvious one is the lack of a mechanism to derive negative literals. Even though several tries to include this possibility into resolution-based reasoners were made, like closed world assumption, program completion and negation as failure, a naive implementation can cause serious troubles since all these approaches introduce non-monotonicity. Due to self-referring rules, the reasoner could run into loops and the closed world assumption can cause inconsistent theories in certain cases. This is depicted more detailed in [Apt and Bol, 1994].

To overcome these problems, a completely new semantics, named the *least fixed point semantics*, for logic programs has been introduced. We are going to follow [Fitting, 1999] and keep working with positive programs only for now. While resolution starts with the query and breaks it down into sub-queries until the halt statement can eventually be derived, the least fixed point semantics starts with the facts. Intuitively the procedure can be described as follows. The facts are initially *true* since they have no premises. Then a rule $H \leftarrow B_1, \ldots, B_n.$ enforces us to set H to *true* whenever all of B_i are *true*. Thus, the set of *true* variables is continuously expanded. The procedure is repeated until no more atoms can be added. Then we say that the *least fixed point* is reached.

Formally we define the operator $\Gamma(P, A)$, where P is a positive logic program and A a set of atoms.

Definition 2.7. The semantics of $\Gamma(P, A)$ is another set of atoms A', s.t. $H \in A'$ iff there exists a ground instance of a rule $H \leftarrow B_1, \ldots, B_n.$ in P and $B_i \in A \; \forall 1 \leq i \leq n$. The semantics of a program P is the least fixed point of this operator, i.e., $lfp(\Gamma(P, \emptyset))$.

Classical Models of Logic Programs

Another semantics for classical logic programs uses a model-theoretical approach. Even though that it was never broadly used in a practical programming environment, it is important for making theoretical comparisons. First we define classical models in contrast to Herbrand models which are introduced in Section 2.2.

Definition 2.8. A model is a a first-order interpretation I s.t. $I \models P$, where $I \models P$ iff $I \models r$ for each ground instance of a rule $r = H_1, \dots H_m \leftarrow B_1, \dots, B_n. \in P$ and $I \models r$ iff $H(r) \cap I \neq \emptyset$ whenever $B(r) \subseteq I$ [Fitting, 1999].

This is demonstrated by Example 2.4.

Example 2.4. Let

$$\begin{aligned} P = \{ & f(a). \\ & g(x) \quad \leftarrow \quad f(x).\} \end{aligned}$$

A classical first-order model can be any interpretation that satisfies all rules of the program. Some models of the program are:

- $I_1 = \{f(a), g(a)\}$
- $I_2 = \{f(a), f(b), f(c), g(a), g(b), g(c)\}$
- $I_3 = \{f(a), g(a), g(b)\}$

Note that there are in general arbitrary many models, where some of them are subsets of others. Intuitively we would say that some of the models seem more natural than others. For instance, I_2 contains constants that never occur in the program and hence it is surprising that it is a model. We come back to this point in the next section.

2.2 Answer-Set Programming

[Gelfond and Lifschitz, 1988] introduced the stable model semantics which is the foundation for answer-set programming. Before we start with a formal definition, we need to introduce the concept of Herbrand models in distinction to classical models as explained in Section 2.1.

In the remaining part of this chapter we will assume that all programs are free of variables. This happens without loss of generality, since programs with variables can be easily transformed into a variable-free version by a procedure called *grounding*. That is, a variable occurring in a rule is simply treated as shortcut for all the rules that can be constructed by replacing the variable by arbitrary domain elements, see Definition 2.9 and 2.10. Note that for function symbols, the grounding is possibly infinite. However, this is not relevant for our purposes since DLV and dlvhex do not support function symbols. For an illustration of grounding see Example 2.5.

Definition 2.9. A term $t \in \mathcal{T}$ is called *ground* iff it contains no variables.

Definition 2.10. The grounding of a ground term (a ground program) is the term (the program) itself. The grounding of an arbitrary term $t \in \mathcal{T}$, denoted as $ground(t)$ (an arbitrary program P, denoted as $ground(P)$), is the set of all variable-free versions that can be obtained by replacing variables by arbitrary ground terms.

Example 2.5. Let's consider the non-ground program

$$\begin{aligned} P = \{ & f(a). \\ & g(b). \\ & f(X) \quad \leftarrow \quad g(X).\} \end{aligned}$$

Then the according grounded program is

$$\begin{aligned} P_{grounded} = \{ & f(a). \\ & g(b). \\ & f(a) \leftarrow g(a). \\ & f(b) \leftarrow g(b).\} \end{aligned}$$

The domain of interest is the set of all ground terms occurring somewhere in the program (this is the Herbrand universe as introduced in the next subsection). For a more detailed description of grounding see [Lifschitz, 2008].

Herbrand Models

Recapitulate Example 2.4, Gelfond and Lifschitz argue that some of the models seem to be somehow more *natural* than others. For instance, I_2 is in fact a model of the above program since $f(a)$ and $g(x) \leftarrow f(x)$. both evaluate to *true* under I_2. However, from a programmer's point of view it would be unexpected to get this model as answer to the program. It is surprising that atoms like $g(b)$ are contained in the model, since the constant symbol b does never occur in the program. Observe that there are infinitely many classical models for a consistent program.

Thus it is necessary to formally define how to select the program answer among this infinite number of models. At this point we need to introduce some definitions following the notation of [Van Emden and Kowalski, 1976].

Definition 2.11. The Herbrand universe of a program P, denoted as $HU(P)$, is the set of all ground terms occurring in P.

Definition 2.12. The Herbrand base $HB(P)$ of a program P is the set of all ground atoms over the Herbrand universe and the predicate symbols in P.

Definition 2.13. A Herbrand interpretation $I \subseteq HB(P)$ is any subset of the Herbrand base.

Definition 2.14. A Herbrand model is a Herbrand interpretation I s.t. $I \models P$.

Now, an obvious solution is to choose Herbrand models as program answer. Since a Herbrand model is a subset of the set of all ground atoms occurring in the program, such a model is in general much more plausible to the programmer since it uses only symbols that actually occur in the program. However, there is still the possibility that the Herbrand model contains ground atoms that are not founded by the intention of the program. Consider Example 2.6.

Example 2.6.

$$\begin{aligned} P = \{ & f(a). \\ & f(x) \leftarrow g(x).\} \end{aligned}$$

In this case, the Herbrand base is $HB(P) = \{f(a), g(a)\}$. The Herbrand models are: $M_1 = \{f(a)\}$ and $M_2 = \{f(a), g(a)\}$. Both of them satisfy all rules of the program and use only symbols that occur in P. However, it seems that $g(a)$ in M_2 is not founded by the rules since it never occurs as fact or consequence of a rule.

To overcome the problem with unfounded atoms, Gelfond and Lifschitz come to the conclusion that one should select the *minimal* Herbrand model (with respect to set-inclusion) as answer to a program. One fundamental result of their paper is the observation that classical logic programs

(i.e., programs without negation) have exactly one minimal Herbrand model, and that model coincides with the smallest fixed point of $\Gamma(P, \emptyset)$ as introduced in Section 2.1. In contrast to that, normal and extended programs can have arbitrary many minimal Herbrand models, see Section 2.2.

Stable Model Semantics

At this point we have introduced the concept of Herbrand models. Now we can switch to logic programs with negations. Such programs are called *normal logic programs* if they use only default negation (not) and *extended logic programs* if they use both default negation and strong negation $\neg$. Note that [Gelfond and Lifschitz, 1988] used the symbol $\neg$ even though they talked about default negation. We will modify the notation and write "not" in order to provide a consistent notation also with the later sections.

Definition 2.15. A normal logic program P consists of rules of the form

$$H \leftarrow B_1, \ldots, B_n.$$

where H is the (possibly empty) head atom and B_i are the body literals (positive or default-negated atoms).

Definition 2.16. Let P be a normal logic program and M be a set of atoms. Then P^M is the program obtained from P by

- discarding all rules $H \leftarrow B_1, \ldots, B_n.$ with a $B_i = \text{not}\, A$ s.t. $A \in M$
- removing all default-negated literals from the bodies of the remaining rules

Then P^M is clearly a positive program and thus has a unique minimal Herbrand model. Later this program was called the *Gelfond-Lifschitz reduct*. If this model coincides with M it is called a *stable set* or *stable model* of P.

Definition 2.17. Let P a program and M a set of atoms. If the unique minimal Herbrand model of P^M coincides with M, it is a stable model of P.

This is demonstrated by examples 2.7, 2.8 and 2.9.

Example 2.7. Consider the following program

$$\begin{aligned} P = \{&p. \\ &q \leftarrow \text{not}\, p. \\ &r \leftarrow \text{not}\, q.\} \end{aligned}$$

Then we can observe:

- $M_1 = \{p, r\}$ is the only stable model since it is the minimal Herbrand model of the reduct $P^{M_1} = \{p., r.\}$
- $M_2 = \{p, q, r\}$ is *not* a stable model since it is *not* the minimal Herbrand model of the reduct $P^{M_2} = \{p.\}$

Example 2.8. The program

$$P = \{p \leftarrow \text{not}\, p.\}$$

has no stable model since the minimal model of $P^{\{p\}}$ is $\emptyset$ ($\neq \{p\}$) and the model of $P^{\emptyset}$ is $\{p\}$ ($\neq \emptyset$)

Example 2.9. The program

$$\begin{aligned} P = \{p &\leftarrow \text{not}\, q. \\ q &\leftarrow \text{not}\, p.\} \end{aligned}$$

has two stable models, namely $M_1 = \{p\}$ and $M_2 = \{q\}$. In contrast, $M_3 = \emptyset$ is not a stable model of the program, since it does not coincide with the minimal model of $P^{\emptyset} = \{p., q.\}$.

Answer-Set Semantics

The stable model semantics has the limitation that negative information cannot be directly expressed but only in form of *negation as failure*. Gelfond and Lifschitz observed this in their fundamental paper [Gelfond and Lifschitz, 1991]:

> A consistent classical theory partitions the set of sentences into three parts: A sentence is either provable, or refutable, or undecidable. A logic program partitions the set of ground queries into only two parts: A query is answered either yes or no.

As explained in Section 2.2, normal logic programs introduce the default-negation *not*. This implements the closed world assumption on predicate level. Each ground atom that does not occur in a candidate for a stable set is assumed to be false. However, there is no way to express that we *know for sure* that something is false.

To overcome this restriction, *extended logic programs* were introduced.

Definition 2.18. A literal L is either an atom A, a strongly-negated atom $\neg A$, a default-negated atom $\text{not}\, A$, or a default- and strongly-negated atom $\text{not}\, \neg A$.

Definition 2.19. An extended logic program P consists of rules of the form

$$H_1 \vee \cdots \vee H_k \leftarrow B_1, \ldots, B_n, \text{not}\, B_{n+1}, \ldots \text{not}\, B_m.$$

where H_i is the possibly empty set of head literals and B_i are the body literals.

We will use the symbol $\neg$ for classical (strong) and "not" for default negation (negation as failure). Further we will denote the set of default-negated body literals $\{B_{n+1}, \ldots, B_m\}$ of a rule r as $B^-(r)$ and the set of non-default-negated (but possibly strongly negated) literals $\{B_1, \ldots, B_n\}$ as $B^+(r)$.

The semantics of extended logic programs is called the *answer-set semantics*, which is an extension of the stable model semantics. In the stable model semantics, the answer of a program was expected to be a set of ground *atoms*. In contrast to that, extended logic programs evaluate to sets of ground *literals*, called *answer sets* [Gelfond and Lifschitz, 1991].

Very similar to Definition 2.16 we can define answer sets as follows.

Definition 2.20. Let P be an extended logic program and M be a set of literals. Then P^M is the program obtained from P by

- discarding all rules r with $B^-(r) \cap M \neq \emptyset$

- removing all default-negated literals from the bodies of the remaining rules

If the unique minimal Herbrand model of P^M coincides with M, it is an answer set of P.

Note that, additionally to the switch from atoms to literals and thus the option of expressing falsity explicitly, the possibility of using disjunctions in the rule heads was added. So called *disjunctive logic programs* were first described by [Przymusinski, 1991] under the stable model semantics and then adopted by Gelfond and Lifschitz for the answer-set semantics.

2.3 HEX Programs

HEX programs are an extension to extended logic programs under the answer-set semantics. The new features include *higher-order atoms* and *external atoms*. We summarize the explanation in [Eiter et al., 2006] and point out the aspects of our interest with respect to later chapters.

Higher-order Logic

Higher-order atoms allow the use of variables instead of predicate names. While the expressions $name(joe)$ and $child(joe, X)$ are ordinary atoms (the first one grounded, the other one not), an expression like $X(joe)$ or $X(Y)$ is a higher-order atom since the predicate name is not fixed. In [Eiter et al., 2006] the importance of this feature is illustrated with the example of ontologies as used in the Semantic Web. Imagine a predicate named $subClassOf(X, Y)$ to express that Y is a specialization of X. X and Y are sets of objects, and whenever an object belongs to Y it also belongs to the (possibly larger) class X. Even though we do not know what X and Y means in details, we can write the following rule:

$$X(Element) \leftarrow subClassOf(X, Y), Y(Element).$$

Quantification over predicate names is called *meta-reasoning* and is not directly expressible in extended logic programs.

The syntax of higher-order atoms is as follows (following the cited paper but using the notation introduced in Section 2.1).

Definition 2.21. A higher-order or ordinary atom is a tuple

$$(Y_0, Y_1, \ldots, Y_n)$$

where Y_i are terms and n is the arity. Y_0 is the predicate name and Y_i for $i \geq 1$ are the arguments. It is an ordinary atom iff Y_0 is constant, otherwise it is a higher-order atom. An equivalent notation is $Y_0(Y_1, \ldots, Y_n)$.

Observe that when grounded, higher-order atoms in fact reduce to first-order atoms. Thus the semantics does not change in comparison to ordinary atoms.

This was just for the sake of completeness. The much more important feature for the later chapters is that of external atoms.

External Atoms

Using extended logic programs, each chunk of information we want to work with needs to be expressed within the logic program using facts and rules. However, sometimes it is useful to work with external information sources, for instance a relational database, an RDF file (as used in the Semantic Web) or an XML file. This is exactly what external atoms allow us.

dlvhex[2] is an open source reasoner for HEX programs. It requires names of external atoms to be prefixed with &, for instance $\&RDF$. External atoms can take arbitrary many constants or predicate names as parameters. As result they deliver the set of all tuples that evaluate to *true* under this predicate.

Definition 2.22. External atoms are expressions of form

$$\&g[A_1, \ldots, A_m](O_1, \ldots, O_n),\ n, m \geq 0,$$

where g is an arbitrary legal external atom name, A_i are the input parameters and O_i are the output parameters. Both are lists of terms, i.e., either constants, variables or predicate names. n is the arity of the external atom.

Intuitively, an external atom has to compute the list of positive output tuples given the input parameters [Eiter et al., 2005]. In case of constants as input parameters, the value is passed directly to the external information source. Predicates are passed as *restricted interpretation* (see below).

The semantics is given as an extension of the answer-set semantics as introduced in Section 2.2. The Herbrand base $HB(P)$ is now the set of all possible ground versions of atoms and external atoms that is constructed by replacing the variables with elements from the set of constants Σ_c that is implicitly given by P.

Since higher-order atoms (i.e., non-external atoms) are just regular atoms after grounding, as mentioned, an interpretation $I \subseteq HB(P)$ is (as usual) a model of such an atom $ground(a)$ iff $a \in I$.

Each external atom $\&g[A_1, \ldots, A_m](O_1, \ldots, O_n)$ has an assigned $(n + m + 1)$-ary semantics function

$$f_{\&g} : 2^{HB(P)} \times (\Sigma_c)^{m+n} \rightarrow \{true, false\}.$$

It is evaluated over the interpretation, which is a subset of the Herbrand base, the input parameters, that are constants in the grounded version, and the output parameters that are again constants.

Definition 2.23. An interpretation I is a model of a ground external atom

$$\&g[y_1, \ldots, y_m](x_1, \ldots, x_n)$$

iff

$$f_{\&g}(I, y_1, \ldots, y_m, x_1, \ldots, x_n) = true$$

This formally defines our intuitive understanding from above. An external atom gets the interpretation and possibly a list of constants and has to return the positive tuples over the Herbrand universe.

Syntax of HEX Programs

Using the previous definitions it is straight forward to put them together and define the syntax of HEX programs. We just reuse the definition of extended logic programs (Definition 2.19). The only thing that needs to be changed is that atoms can now be either higher-order or external.

Definition 2.24. [Eiter et al., 2006] A HEX program is a set of rules of the following form:

$$H_1 \vee \cdots \vee H_k \leftarrow B_1, \ldots, B_n, \text{not } B_{n+1}, \ldots \text{not } B_m.$$

where H_i are higher-order and B_i are either higher-order or external literals.

[2] `http://www.kr.tuwien.ac.at/research/systems/dlvhex`

Semantics of HEX Programs

For defining the semantics of HEX programs, [Eiter et al., 2005] use the concept of the *FLP-reduct* rather than the *Geldfond-Lifschitz reduct* which was used in the definition of ordinary answer-set semantics (see Section 2.2).

Definition 2.25. Let P be a program and $I \subseteq HB(P)$ an interpretation. Then the FLP-reduct of P with respect to I, denoted as fP^I, is defined as the set of all ground rules that are satisfied by I, i.e. $fP^I = \{r \in ground(P) \mid I \models r\}$ [Faber et al., 2010]. As usual, $I \models r$ iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$.

Definition 2.26. An interpretation I is an answer set of program P iff I is a subset-minimal model of fP^I.

In contrast to the traditional Gelfond-Lifschitz reduct, the FLP-reduct ensures minimality of answer sets and thus allows a more elegant definition of the semantics of programs with aggregates. For more details see [Eiter et al., 2005].

Implementation

We come back to the syntax of external atoms. Recapture that the input parameters can either be constants, variables or predicate names, though in theory they are all equivalent since they are all reduced to constants from Σ_c when the program is grounded. In practice, this difference is of importance due to an optimization strategy.

Actually, dlvhex will not pass the complete interpretation I to an external atom's implementation since most atoms do not need the complete information in order to compute their result. Instead it will extract the predicate names that are explicitly mentioned in the input list and will then pass a *reduced interpretation*, that contains only those atoms that are built upon the predicates in the input list. That is, the programmer needs to define explicitly what predicates the external atom is actually going to use. This prevents that huge data structures are passed even if only very few predicates are evaluated.

External atoms are implemented as shared libraries that are dynamically linked to dlvhex at runtime. This enables the user to write custom external atoms without recompiling the main program. Each library can define arbitrary many external atoms that are registered within dlvhex on startup through a standardized interface. Such libraries are written in C++ and installed in dlvhex' plugin directory (or in a custom one, that is explicitly mentioned in the command-line arguments). The development kit contains a standardized interface that needs to be implemented for each external atom.

2.4 Requirements

Now we have summarized the history of logic programming and introduced the basic notations and principles. Before we continue with the development of extensions to this formalisms, we point out the relevant aspects of the last sections.

In the remaining part of the thesis we will solely work with the extended answer-set semantics for HEX programs (Section 2.3). Especially the new feature of external atoms will be exploited. In the practical part, dlvhex is the reasoner we use.

While the main topic of the thesis is *belief merging*, i.e., using knowledge from several sources, there is the need to introduce the concept of *nested* HEX *programs* first. In contrast to DLV, dlvhex enables the user to use external sources of computation, but it is still tricky to "call"

sub-procedures in the same way as this is usual in procedural programming. This the the topic of the next chapter.

If it's green, it's biology, If it stinks, it's chemistry, If it has numbers it's math, If it doesn't work, it's technology

Anonymous

Chapter 3

Nested HEX Programs

This chapter introduces the concept of *nested* HEX *programs* since this will be the basis for the development of the belief merging framework in Chapter 4.

As we have seen in the last chapter, many different formalisms for logic programming have been developed in the last decades. The historical summary has shown that the kind of rules of which programs are composed of has continuously been generalized. However, even though HEX programs allow the import of knowledge from external sources, it is still impossible to partition a huge program into several sub-procedures that call each other in the same way as this is done in procedural programming.

Informally, the term *nested* HEX *programs* expresses exactly this feature. It allows the programmer to *call* other programs, let them compute their answer sets and retrieve them in the calling program. Then computation of the host program can continue. This can not directly be done in pure logic programs without special plugins, when external atoms are implemented in C++ libraries and are therefore procedural rather than declarative.

We will show how to develop an external atom that internally returns from procedural code to another instance of a declarative reasoner. With this mechanism, we stay within declarative programming but can compute results of subprograms independently, which is the basis for *modular programming*. This has already been studied by [Veith et al., 1997].

As a short lookahead, the intention of the development of such a plugin is the merging of several belief bases. We will then be able to treat each belief source as independent module (each is implemented as HEX program, possibly with further evaluations of external atoms) that can be called. Finally we retrieve its information and merge them within a superior program to generate the final result.

3.1 Motivation

When we think of classical procedural programming, it is trivial to write a program that calls another program and process its output and return value. One could say this kind of code reuse is one of the principles of procedural programming, that reduces development time (e.g., by using third-party libraries), reduces the length of the sourcecode and, last but not least, makes code human-readable. To understand this, just imagine a typical application of any kind and

with today's dimensions written in just one giant method. Nobody could read or debug such an application.

The same kind of code reuse is also well-known in *functional programming*, another declarative paradigm. However, in logic programming calling other programs is very tricky. Even though DLV, dlvhex and other reasoners enable the user to partition a program in several files, they are still merged into *one* logical set of rules when the interpreter starts up.

This is not an adequate feature when talking about calling subprograms, as the following examples illustrate.

Resrictions of Logic Programs

As a first motivating example see 3.1. The program *binom* calls subprogram *faculty* in order to compute the binomial coefficient of numbers n and k.

Example 3.1.
Procedural program:

```
int binom(int n, int k){
        return faculty(n) / ( faculty(n - k) * faculty(k) );
}

int faculty(int n){
        if (n > 1) {
                return n * faculty(n - 1);
        }else{
                return n;
        }
}
```

HEX program:

$$\begin{aligned} P = \{ binom(N,K,R) \quad \leftarrow \quad & faculty(N,RN), \\ & N = NmK + K, faculty(NmK, RNmK), \\ & faculty(K,RK), \\ & D = RNmK * RK, RN = R * D. \} \end{aligned}$$

It is relatively easy to translate this program into a HEX program. If we assume that we have a predicate $faculty(N,R)$, that is *true* if and only if R is equal to $N!$, we can define predicate $binom(N,K,R)$ as shown. We first compute the faculty of N, K and $N-K$ and then use the results RN, RK and $RNmK$ for computing the binomial coefficient. $binom(N,K,R)$ holds if and only if $R = \binom{N}{K}$.

However, calling the "subprocedure" *faculty* is only possible because of the fact that its result is a single value. If a subprogram delivers several results, the translation can be tough or even impossible. For instance, the next example shows a program that randomly invites 1 or 2 relatives of *john*.

Example 3.2. This program was taken from [Eiter et al., 2005] who used it to illustrate the semantics of HEX programs. It randomly selects 1 or 2 of John's relatives and invites them by the use of the external atom &*reach* and exploiting higher-order capabilities in one rule. Assume that &*reach* is defined such that it is *true* iff there is a connection between some person X and

john).

$$
\begin{aligned}
S = \{ & subRelation(brotherOf, relativeOf). \\
& brotherOf(john, al). \\
& relativeOf(john, joe). \\
& brotherOf(al, mick). \\
& invites(john, X) \vee skip(X) \leftarrow X <> john, \\
& \qquad \&reach[relativeOf, john](X); \\
& R(X, Y) \leftarrow subRelation(P, R), P(X, Y). \\
& \leftarrow \&degs[invites](Min, Max), \\
& \qquad Min < 1. \\
& \leftarrow \&degs[invites](Min, Max), \\
& \qquad Max > 2. \}
\end{aligned}
$$

Each selection of relatives forms an answer set. And this the important thing to observe. We do not have a binary predicate like $selected(X, Y)$, that is *true* for all the possibilities (with a dummy value if just one person is selected), but the choices are returned in *separate* answer sets.

Now imagine we are not interested in the selection itself but in the number of possible choices that satisfy the constraints. In other words, we want to count the answer sets of the program.

In procedural programming solving this task is straightforward. We just call the subroutine that returns the pairs as array, linked list, or any other suitable data structure, and count the number of elements (where the details are programming language dependent). For an example implementation see Example 3.3.

Example 3.3. A procedural implementation of the invitation program.

```
// Assume that Selection is a structure with the members
//   count and persons[]
// where count is either 1 or 2 and person is an array of
// one or two names
Selection[] getSuggestions(){
        Pair[] result = ... ;
                // some algorithm to compute the possible
                // selections among john's relatives
        return result;
}

int countChoices(){
        Selection[] possibilities = getSuggestions();
        // Exact syntax depends on the programming language
        return possibilities.length;
}
```

In case of a logic program we run into troubles. Since there is no support for actual subroutine calls, all we can do is to *add* rules to the program *if* - and this is the point - we treat the given program as black box. The assumption of a program as black box is very natural and reasonable, since in case of third-party sources we probably do not know the internal details, and in particular we can not modify them.

To summarize, we are given a program like the one in Example 3.2, denoted as S (for *suggestions*), and are only allowed to add a set of rules R. In this case, we simply cannot determine the number of suggestions since this would require reasoning on the level of *sets of*

answer sets. However, the point is that while one answer set of $S \cup R$ is computed (for instance, that where *al* and *mick* are invited), we can only access the atoms within *this* answer set. But we can not find out how many *other* answer sets $S \cup R$ has and how many we have in total or what they look like. Thus we cannot derive an atom like $invitationCount(X)$.

Overcoming the Limitations

As we see, in declarative programming it is far from trivial to call subprograms, let them be computed *independently* and use their results. Rather the *"calling program"* (subsequently called *host program*) and the *"callee"* are always coupled together very tightly. Thus, what we desire is an easy to use mechanism for subprogram calling.

In the following sections the semantics for a reasoner on the *sets-of-answer-sets-level* will be developed. Before we do this we take a closer look at the structure of answer sets in Section 3.2.

3.2 Results of HEX Programs

The result of an extended logic program is a set of answer sets. There can be one answer set (as for instance in case of Horn programs), none (in case of contradictions) or multiple (by using disjunction or default-negation). Consider for instance the example programs 3.4, 3.5 and 3.6.

Example 3.4. A program with one answer set: $\{p, q, r\}$

$$\begin{aligned} P_1 = \{ & p. \\ & q. \\ & r \leftarrow p. \} \end{aligned}$$

Example 3.5. A program with no answer sets.

$$P_3 = \{p \leftarrow \text{not } p.\}$$

Example 3.6. A program with two answer sets: $\{p\}$ and $\{q\}$

$$\begin{aligned} P_3 = \{ & p \leftarrow \text{not } q. \\ & q \leftarrow \text{not } p. \} \end{aligned}$$

Each answer set is in turn a set of literals. Finally, a literal is in general a (positive or strongly negated) predicate that depends on arbitrary many terms. Figure 3.1 summarizes the relationships.

3.3 Requirements

When talking about *calling programs*, the first question to answer is *which* program to call. Basically there are two possibilities for the location of such programs. Either they are directly embedded within the host program, or they are stored completely separately in a different file.

Embedded programs are useful when both the calling and the called program are tightly coupled and are developed together. This is very similar to a procedural program that consists of several subprograms. Usually such applications are split up in modules where each of them fulfills a certain subtask. This makes the program more readable.

Even though the called procedure comes from the same developer, it cannot simply be included in form of additional rules. As we have seen, the semantics does not allow us to investigate

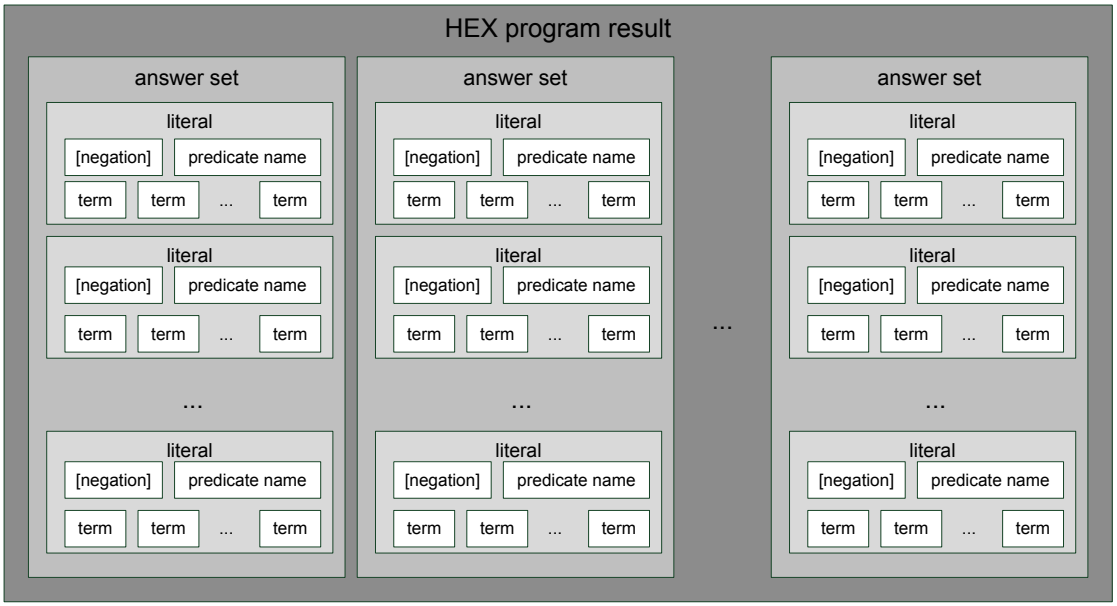


Figure 3.1: Structure of HEX program results

the result independently from the host program since we can always look only into the *current* answer set. A solution to this problem is to embed the subprogram as string literal in form of a constant into the host program.

The second variant occurs when we call third-party programs or programs that were developed independently from the caller. In this case we cannot include the callee directly within the host, or at least, this is disadvantageous since we have to maintain a *copy* of the code that needs to be updated when the original program is modified. In this case it seems to be the better choice to include just the *path* to the external program, just like a procedural function would contain the *name* of the callee. For now it is sufficient to consider paths as abstract references. We will take a look at the implementation below.

After we have called another program we want to access and investigate its results. Thus we need to define external atoms that enable us to request all of the elements listed in Figure 3.1 in the previous section. First, we need to request the answer sets of a program. Next we want to look into the answer sets and retrieve its literals. Finally, we want to retrieve the sign (positive or strongly negated) and the arguments of a literal. If a set of functions is provided which allow all these lookups, it is possible to talk about HEX programs within other ones.

3.4 Formal Semantics

We have discussed the requirements for a reasonable interface for sub-procedure calling. This fairly informal description is now converted into a definition of appropriate external atoms.

Let $\mathscr{P}$ be the set of all HEX programs over signature $\Sigma = (\Sigma_v, \Sigma_p, \Sigma_c)$ as introduced in Section 2.1.

From Programs to Answer Sets

First we need a predicate that allows us to look into the answer of a program and find out how many (and subsequently: which) answer sets it has. Remember Example 3.2 where it would be sufficient to know the number of answer sets.

We define a predicate *answersets* which maps programs to sets of answer sets, i.e., sets of sets of literals.

Definition 3.1. The external atom *answersets* is defined as follows:

$$answersets : \mathscr{P} \rightarrow 2^{2^{Lit}}$$

where the set of literals is defined as

$$Lit = \{p(t_1, \ldots, t_n), \neg p(t_1, \ldots, t_n) | p[n] \in \Sigma_p, t_i \in \mathcal{T}\}$$

This is illustrated with an example.

Example 3.7. Let $P = \{p \vee q.r \leftarrow p.s \leftarrow q.\}$. Then $answersets(P) = \{\{p, r\}, \{q, s\}\}$.

Predicates within Answer Sets

Next we define an atom that maps answer sets to tuples of all the predicate names that occur within it, paired with their arity.

Definition 3.2. The external atom *predicates* is defined as follows:

$$predicates : 2^{Lit} \rightarrow 2^{(\Sigma_p \cup \neg\Sigma_p) \times \mathbb{N}}$$

where $\neg\Sigma_p$ is our abbreviation for the set of all strongly-negated predicate names

This is again demonstrated with an example.

Example 3.8. Let $AS_2 = \{node(a), node(b), edge(a, b), leaf(b), \neg leaf(a)\}$.
Then, $predicates(AS_2) = \{(node, 1), (edge, 2), (leaf, 1)\}$.

Note that in case a negative literal occurs in the answer set, the result will nevertheless contain the positive version of the predicate name. We come back to the sign in the next subsection.

Predicates have Arguments

Finally, a predicate has up to n arguments (constants). Thus we need one more atom for retrieving them.

Definition 3.3. The external atom *arguments* is defined as follows:

$$arguments : 2^{Lit} \times \Sigma_p \rightarrow 2^{\mathbb{N} \times \mathbb{N}^s \times \mathcal{U}}$$

where $\mathbb{N}$ is the set of natural numbers and $\mathbb{N}^s$ is $\mathbb{N}$ extended with a special symbol s (see below)

Intuitively, *arguments* maps a given tuple of an answer set and a predicate name (let n be its arity) to a subset of triples consisting of natural number, one element of $\mathbb{N}^s$ (which can be a natural number or s) and a domain element.

To understand the atom, consider the following example.

Example 3.9. Let $AS_3 = \{node(a), node(b), node(c), edge(a,b), edge(c,b)\}$
Then

$$arguments(AS_3, edge) = \{(\mathbf{0}, s, 0), (\mathbf{0}, 0, a), (\mathbf{0}, 1, b), (\underline{1}, s, 0), (\underline{1}, 0, c), (\underline{1}, 1, b)\}$$

Since one predicate name can occur multiple times within an answer set, as this is the case for predicates *node* and *edge* in the example, the first natural number is a running index. It is initially 0 and incremented for each further occurrence of the same predicate name. Thus, all triples with the same value at the first parameter position describe one occurrence. In the example, all 0-triples are **bold** and all 1-triples are underlined. Since the index ranges from 0 to 1, one can conclude that *edge* occurs twice in AS_3.

The second natural number represents the parameter position that can be anything $\in \{s\} \cup \{0, 1, \ldots, n\}$, where the special symbol s represents the *sign* of the literal (0 for positive, 1 for strongly-negated). Finally, the domain element is the value at this position.

For the first occurrence of *edge* (index 0; bold) within AS_3, the sign is positive (0), its 0-th parameter is a and its 1-st one is b. Therefore we have the atoms, $(\mathbf{0}, s, 0)$ (since it is positive), $(\mathbf{0}, 0, a)$ and $(\mathbf{0}, 1, b)$. The second occurrence (index 1) is also positive, its 0-th parameter is c and its 1-st one is b, leading to atoms $(\underline{1}, s, 0)$, $(\underline{1}, 0, c)$ and $(\underline{1}, 1, b)$.

Note: Since answer sets are especially *sets*, the numeric value of the first parameter is irrelevant because the order does not matter. The only thing of interest is that it is different for each occurrence. It enables the user to know which of the parameter descriptions belong together. Without the running index, one could not determine if the answer set contains $edge(a, b)$ and $edge(c, b)$ or $edge(a, c)$ and $edge(c, b)$.

Using these functions, one can access all elements of the results of a HEX program from $\mathscr{P}$. However, the details about the bootstrapping problem, namely how to execute a program to get its answer sets, are still open. This leads to the section about implementation.

3.5 Implementation

In the last sections we introduced the structure of a HEX program result and the requirements we have in order to access all of their attributes. We further have shown that this can be done using the 3 functions *answersets*, *predicates* and *arguments* and formally defined them.

Now we consider how we can actually implement this functionality as an extension to dlvhex. As described in Section 2.3, one of the two main enhancements of dlvhex in comparison to DLV is the possibility to write plugins that implement external atoms.

First we will show how the bootstrapping problem can be solved. That is, how external programs can actually be called. Then we will depict the implementation of the described external atoms schematically.

Figure 3.2 shows the architecture of the mergingplugin. Basically it consists of a set of external atoms for nested HEX program execution as well as a cache for HEX answers.

Calling External Programs

The function *answersets* maps programs to answer sets. But how do we get the *program* and, even more interesting, how can we represent a program within another one?

It is clear that we cannot simply insert the callee into the caller, since this would cause dlvhex to execute the resulting file as one big program, as explained in Section 3.1. But what we can do is to include the path to an external program in a string literal (a constant in terms of procedural programming), pass it to an external atom, let this program be loaded and retrieve a symbolic handle to this program.

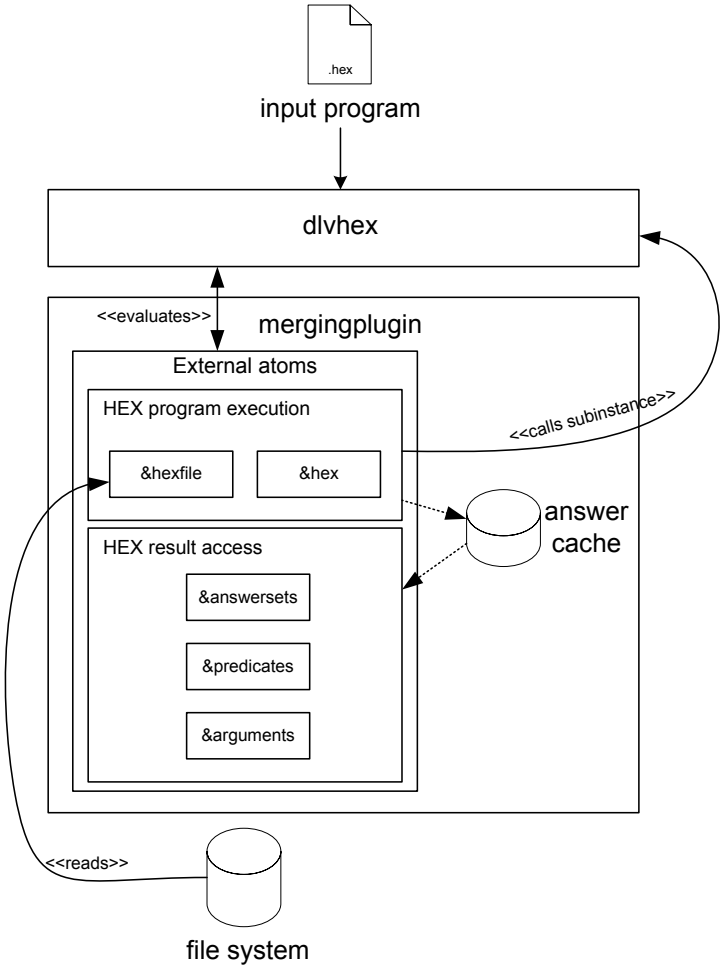


Figure 3.2: mergingplugin internals

This is exactly what the atom &*hexfile* of the plugin does. The following example demonstrates the functionality.

Example 3.10.

$$P = \{handle(H) \quad \leftarrow \quad \&hexfile[\text{“}externalprogram.hex\text{”}, \text{“”}](H).\}$$

The code snippet will execute the program named *externalprogram.hex*, writes its result into the internal cache and returns a handle (ignore the second parameter, the empty string, for now). A handle is just a symbolic integer value to a certain cache entry. That is, the numeric value is not of interest since it does not encode any information. But it is unique for each external program and can be used later on to refer to the result again. This is just as that the numeric values of pointers in low-level programming languages are irrelevant, as long as we know that they point to a certain data structure. The example program will have one answer set, namely $\{handle(0)\}$. The fact that this result is independent from the contents of *externalprogram.hex* demonstrates that 0 is actually just a *handle* and not a set of answer sets.

Formally, &*hexfile* is a mapping from strings to natural numbers:

$$\&hexfile : String \rightarrow \mathbb{N}$$

Now that we have an external atom for executing programs stored in files, it suggests itself to provide another external atom that can execute HEX programs embedded directly within the calling program in form of a string. The following snippet illustrates this.

Example 3.11.

$$P = \{handle(H) \quad \leftarrow \quad \&hex[\text{“}a.\ b.\ c \leftarrow a.\text{”}, \text{“”}](H).\}$$

Here, the program $P_{emb} = \{a., b., c \leftarrow a.\}$ is directly passed to the external atom &*hex*. The result is again a handle to the program's set of answer sets. In this example it will again be 0.

Now consider another example:

Example 3.12.

$$\begin{aligned} P = \{handle1(H) \quad &\leftarrow \quad \&hexfile[\text{“}externalprogram.hex\text{”}, \text{“”}](H). \\ handle2(H) \quad &\leftarrow \quad \&hexfile[\text{“}externalprogram.hex\text{”}, \text{“”}](H). \\ handle3(H) \quad &\leftarrow \quad \&hex[\text{“}a.\ b.\text{”}, \text{“”}](H).\} \end{aligned}$$

The answer will either be

$$\{handle1(0), handle2(0), handle3(1)\}$$

or

$$\{handle1(1), handle2(1), handle3(0)\}$$

depending on the order in which the external programs are executed (there are no guarantees about this and it is irrelevant anyway). However, $handle1(X)$ and $handle2(X)$ will for sure contain the same value for X. Due to the caching mechanism of &*hex* and &*hexfile*, it is noticed that the result of *externalprogram.hex* is referred to twice. But in fact it is executed only once due to memory and runtime reasons. Thus, both calls will return the same handle to the according cache entry. In contrast, $handle3(Y)$ will have a $Y \neq X$, since the embedded program $\{a., b.\}$ is (possibly) different from the program *externalprogram.hex*.

Back to the second parameter. Both &*hexfile* and &*hex* expect two strings: the first one is either a path to a program or the program itself, the second one is the string containing the command-line arguments (if any) that shall be passed to dlvhex when the external program is executed.

Note that the atom &*hex* provides a quoting mechanism for the case that the embedded program contains quotation marks itself. More about this can be found in the documentation of mergingplugin.

Accessing Answer Sets

To summarize, at this point we have executed an external program and retrieved a handle to its answer that is stored in the plugin's cache. But we have no idea what this answer actually looks like.

Recall the formal semantics of $answersets : \mathscr{P} \rightarrow 2^{2^{Lit}}$. We will adjust this definition to make it suitable for being implemented. First, we said program results are represented by handles in form of natural numbers. Second, the function needs to return sets of answer sets. However,

since logic programs do not support custom data structures (like classes in Java or structs in C) it is very inconvenient to actually work with answer sets. We can only choose between strings and constants (i.e., unquoted strings) since this is everything we have in dlvhex. Thus it would be necessary to encode answer sets in strings in order to return them from or pass them to external atoms. This is very expensive both in memory and time since a lot of string copy and answer set parsing operations were necessary.

Therefore we will again work with handles.

$$answersets : \mathbb{N} \to 2^{\mathbb{N}}$$

A handle to a program answer is mapped to a set of handles to its answer sets. This is illustrated with an example.

Example 3.13.

$$\begin{aligned} P = \{ ash(PH, AH) \quad \leftarrow \quad & \&hex[\text{“}a \vee b.\text{”}, \text{“”}](PH), \\ & \&answersets[PH](AH).\} \end{aligned}$$

The embedded program has two answer sets: $\{a\}$ and $\{b\}$. $\&hex$ will return a handle to the program's answer, namely $PH = 0$ (*program handle*). This handle is then passed to $\&answersets$, which returns a *list* of answer set handles, in this case 0 and 1 (due to the fact that the program has two answer sets). The final result of the above program is thus one answer set $\{ash(0,0), ash(0,1)\}$. Note that the answer sets handles start with 0 for each program. Only a pair of program handle and answer set handle uniquely identifies an answer set.

Looking into Answer Sets

In the last section we developed all the machinery that we need in order to solve the problem we introduced as motivating Example 3.1.

Assume that the program *suggestions.hex* returns either persons or tuples of persons that are invited, one suggestion per answer set. Now we can easily write another program that counts the number of total possibilities:

Example 3.14.

$$\begin{aligned} P = \{ as(AH) \quad \leftarrow \quad & \&hexfile[\text{“}suggestions.hex\text{”}, \text{“”}](PH), \\ & \&answersets[PH](AH). \\ number(D) \quad \leftarrow \quad & as(C), D = C + 1, \text{not } as(D).\} \end{aligned}$$

The program computes D, such that $D - 1$ is the largest existing handle to an answer set (due to index origin 0). Thus, D is the number of answer sets of *suggestions.hex*.

However, we still treat answer sets of nested programs as black box. Now we go a step further and look into them. In other words, we implement the function *predicates*. Recall the definition: an answer set is mapped onto a set of tuples of predicate names and natural numbers (arities). The according definition for our implementation is:

$$predicates : \mathbb{N} \to 2^{String \times \mathbb{N}}$$

We illustrate the usage of this external atom with an example.

Example 3.15. This program extracts all the predicates together with their arities, that occur in the answer of the embedded program.

$$\begin{aligned} P = \{h(PH, AH) &\leftarrow \&hex[\text{“}node(a).\ node(b).\ edge(a,b).\text{”}, \text{“”}](PH), \\ &\quad \&answersets[PH](AH). \\ preds(P, A) &\leftarrow h(PH, AH), \&predicates[PH, AH](P, A).\} \end{aligned}$$

The only answer set of this program is

$$\{h(0,0), preds(node, 1), preds(edge, 2)\}$$

since the embedded program has one answer set using the unary predicate *node* and the binary relation *edge*.

Retrieving Predicate Arguments

The example in the last section demonstrated that we can already determine the predicates that are used in a certain answer set. But we still don't know the parameters of these predicates. This leads to the last function: *arguments*. In theory (see Section 3.4), this function maps a predicate name to triples of one integer, one integer or the special sign character s and one string. And in this case, this is also the practical definition to be implemented. This is again best demonstrated with an example.

Example 3.16.

$$\begin{aligned} P = \{handles(PH, AH) &\leftarrow \&hex[\text{“}node(a).\ node(b).\ node(c).\ edge(a,b). \\ &\quad edge(c,a).\text{”}, \text{“”}](PH), \&answersets[PH](AH). \\ edge(Value0, Value1) &\leftarrow handles(PH, AH), \\ &\quad \&arguments[PH, AH, \text{“}edge\text{”}] \\ &\qquad (RunningIndex, 0, Value0), \\ &\quad \&arguments[PH, AH, \text{“}edge\text{”}] \\ &\qquad (RunningIndex, 1, Value1).\} \end{aligned}$$

The only answer set of this program is

$$\{handles(0,0), edge(a,b), edge(c,a)\}$$

The embedded program delivers one answer set. Thus we retrieve $handles(0,0)$. Using the external atom $\&arguments$ we look into it. We pass the handle to the program's answer as well as the handle to its first (and only) answer set. We further pass a predicate name, in this case “*edge*” (we assume that we *know* that the nested program uses this predicate for reasons of simplicity, we could also determine the used predicates by using $\&predicates$).

The result is a triple of a running index (a different value for each occurrence of the *edge* predicate), an argument position index (in this case 0 for the start vertex of the edge and 1 for the other one) and the actual argument value. Note that we use the variable *RunningIndex* in both $\&arguments$ calls. This makes sure that *Value0* and *Value1* are actually endpoints of the *same* edge.

To check the sign S of a literal we would need to query

$$\&arguments[PH, AH, Pred](RunningIndex, s, S)$$

exploiting the special constant symbol s for the second output parameter, but in the example we just assume that we know that *edge* only occurs positively.

This concludes our implementation of nested HEX programs. We are now able to execute nested programs and access everything that is stored within their answer sets. This will be an important foundation for the framework developed in the remaining part of the thesis.

Whenever a theory appears to
you as the only possible one, take
this as a sign that you have
neither understood the theory
nor the problem which it was
intended to solve.

Karl Popper

Chapter 4

Formal Development of the Framework

In Chapter 2 we summarized the history of logic programming and introduced the theoretical foundations of modern answer-set programming. Chapter 3 was an excursus to nested HEX programs since this concept will be of special importance for later sections.

Now we come to the main topic of this thesis. At the beginning we will work out the requirements we have to the framework using a few illustrating examples. Finally, a formal task definition will be given. Subsequently we will show how to solve that task using nested HEX programs and the new concept of *merging operators*. This will again be demonstrated with examples.

At the end of the chapter the foundations for the implementation in Chapter 5 will be laid down.

4.1 Motivation

Nowadays many fields of science work with huge amounts of data provided by third parties. That can be for instance protein databases in biomedical applications (e.g., SWISSPROT[1], like depicted in [Bairoch and Apweiler, 1997]), ontologies, XML sources in the Semantic Web (see for instance [Decker et al., 2000]) or collections of scientific literature.

In many cases there exist several different sources for a certain domain. The information provided by these *belief bases*, as we call them from now on, is often similar but not entirely equivalent. Such *inconsistencies* arise from several influence factors. First, if different scientific groups work on similar projects, they will often come to similar but not equal conclusions (consider for instance experimental studies in natural sciences). Second, several versions of the same belief base may exist that are maintained by different people. Further reasons can be rounding errors, different opinions, incomplete information due to industrial secrets or simply human errors.

Thus we have a couple of belief bases that overlap partially, complement one another but may also be contradicting in some details. The question is then which of the sources we trust. Clearly the answer depends on the application in mind. Sometimes the reliability of one source is higher than of the others. In other cases we could try to extract as much from the single

[1] `http://expasy.org/sprot/`

sources as possible without introducing contradictions. The conflict resolution strategies can be very different.

A system for semiautomatic belief merging is strongly desirable. Otherwise it is necessary to investigate the sources by hand, extract relevant information and decide which of the data units (which can be for instance tuples in relational databases or entities in ontologies) to include in the final merged database.

What follows is an introduction of belief merging. First we introduce the topic using a model-theoretic approach. In Section 4.5, we switch our point of view and consider answer sets.

4.2 Belief Merging

We now take a closer look at the term *belief merging* and separate it from *belief revision*. In belief revision, an agent with its current beliefs is faced with new information that needs to be incorporated into its belief base [Gabbay et al., 2009]. If the new information is consistent with its knowledge it can be simply added. Otherwise the agent needs to adapt its beliefs. Alchourròn, Gärdenfors and Makinson have discovered some (mostly intuitively understandable) reasonable requirements we expect from a useful revision operator. Today they are called *AGM postulates* [Alchourròn et al., 1985].

Belief Revision

The formal definition is as follows, where KB is some belief base, $\circ$ is a revision operator and ϕ and ψ are first-order formulas to be added. Cn denotes the deductive closure.

$K^\circ 1$) $KB \circ A$ is a belief set

$K^\circ 2$) $\phi \in K \circ \phi$

$K^\circ 3$) $K \circ \phi \subseteq Cn(K \cup \{\phi\})$

$K^\circ 4$) If $\neg\phi \notin KB$, then $Cn(KB \cup \{\phi\}) \subseteq KB \circ A$

$K^\circ 5$) $KB \circ \phi = \bot$ only if $\phi \equiv \bot$

$K^\circ 6$) If $\phi \equiv \psi$, then $KB \circ \phi \equiv KB \circ \psi$

$K^\circ 7$) $KB \circ (\phi \wedge \psi) \subseteq Cn((KB \circ \phi) \cup \{\psi\})$

$K^\circ 8$) If $\neg\psi \notin KB \circ \phi$, then $Cn((KB \circ \phi) \cup \{\psi\}) \subseteq KB \circ (\phi \wedge \psi)$

To summarize them informally, they basically express that a belief base should be changed as little as possible such that it is consistent with the new information. Then the new belief (a formula) can be simply added.

In other words, old beliefs are only given up, if this is absolutely necessary in order to add the new belief without introducing contradictions. However, this also implies that the new belief has a higher priority than the existing formulas in the base.

Belief revision operators are very often defined in such a way that they select a subset-maximal set of formulas, that is contained in KB and does not imply $\neg\phi$. Observe that this set is not unique in general. Therefore the revision operators differ in the way how this set is selected. Some of them take for instance the intersection (*full meet* or *partial meet contraction*), others take just one of the possibilities (*maxichoice contraction*) cf. [Gabbay et al., 2009].

Separation from Belief Merging

In contrast to the incorporation of new information into an existing knowledge base, with the goal to keep the result consistent, belief merging deals with the aggregation of multiple finite sets of information sources into a single one.

Following the principle of minimal change like introduced for belief revision, one property we can expect from a reasonable operator is that the models of the resulting belief base are somehow *similar* to the models of the input belief bases. In the literature, this was formally defined as follows. Let $K = \{KB_1, \ldots, KB_n\}$ be the input belief bases with models $Mod(KB_i)$. Then the distance D^d between any interpretation I and the set of belief bases K is:

$$D^d(I, K) = D(d(I, KB_1), \ldots, d(I, KB_n)),$$

where $D^d : \mathbb{R}^n \to \mathbb{R}$ is called the *aggregate function* that summarizes the distances of I to the different input belief bases into a single value (this can, for instance, include a weighting process).

Let $\mathcal{M}$ be the domain of propositional interpretations and let $\underline{d} : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$ be Dalal's distance (as introduced in [Dalal, 1988]) between two interpretations. Then the distance d (not underlined!) of interpretation I to a single belief base KB_i is defined as

$$d(I, KB_i) = \min_{J \in Mod(KB_i)} \underline{d}(I, J)$$

with the side constraints $\underline{d}(X, Y) = \underline{d}(Y, X)$ and $\underline{d}(X, Y) = 0$ iff $X = Y$.

Informally, this states that the difference between I and a knowledge base KB_i is computed upon the *best fitting* model of KB_i. The exact (symmetric) distance between I and (some other interpretation) J is expressed by a real number, where equivalent interpretations have distance 0.

With these definitions it is easy to select the merged belief base such that its (only) model has a minimal value under D.

Definition 4.1. A majoritarian merging operator $\circ^n$ for a set of belief bases K is given by

$$\circ^n(K) = form(\arg\min_{I \not\models \bot} D^d(I, K)),$$

where $form(I)$ is a new belief bases such that $Mod(form(I)) = \{I\}$

We can even extend this definition such that the result of the operator is not only similar to the sources, but also respects some *side constraints* (also called *integrity constraints*):

Definition 4.2. A majoritarian merging operator $\circ^n$ for a set of belief bases K that respects side constraints C is given by

$$\circ^n(K, C) = form(\arg\min_{I : I \not\models \bot, I \models C} D^d(I, K))$$

where $form(I)$ is a new belief bases such that $Mod(form(I)) = \{I\}$

Clearly these definitions define a concrete merging operator by no means at all. There is still much room for different variants that are strongly application dependent. But they force operators to satisfy some reasonable constraints. What remains to be defined for merging tasks are the details of D and $\underline{d}$. At this point we give a first example for a merging operator.

Example 4.1. According to [Gabbay et al., 2009], the most widely used operator just uses the sum of the distances as D, i.e.,:

$$D^d(I, K) = \sum_i d(I, KB_i)$$

and the number of differing propositional letters in I and J for $\underline{d}(I, J)$, i.e., the Hamming distance:

$$\underline{d}(I, J) = |\{p \in (I \cup J) \mid I(p) \neq J(p)\}|$$

In contrast to belief revision we now *don't* have new information which has absolute priority. We just seek for a compromise that is as near to the input belief bases as possible.

4.3 Sources of Incompatibility

If we take a closer look at inconsistencies we can observe that there exist in fact several types on different levels. Some of them seem to be more direct and thus more obvious whereas others require semantic analysis and interpretation of the data.

What follows is an investigation of these types of inconsistencies.

Syntactic Compatibility

What we skipped until now is the fact that in practical applications, where we do not deal with formulas but with concrete technologies, belief bases are often not even compatible with respect to their knowledge representation formalism. Thus, before we can talk about *contradictions* in logical sense, we have to make sure that belief bases are compatible. That is, if one belief base is a relational database and the other one an XML file, we can not say if they contradict each other since a kind of conversion must be done first.

This problem arises on several different levels. Not only knowledge representation formalisms can be incompatible but also design decisions within a belief base. Imagine a relational database for a real world applications. Then there are in general many different possibilities how the schema can look like, and each of them is correct. It is up to the database designer to select one. And one step further, even if belief bases are built upon the same EER model they can be slightly differing. Imagine two customer databases, one storing the currency values in US-Dollars and the other one in Euros (the EER model does not contain any information about this detail). Then the values $1.369,10$ and $1.000,00$ may look contradicting at first glance, but in fact they are perfectly compatible (assuming a currency rate of 1 EUR $= 1,3691$ USD, February 05, 2010).

Another example for a seeming incompatibility on this level is that of different date formats. The attribute values 04.07.2010 and 2010-07-04 are equivalent, if one knows that the first date is in European and the latter one in ISO notation.

We will call all these effects *syntactic variants*. To summarize, it occurs on the following levels: knowledge representation formalism, data model and data types.

In the context of relational databases a straightforward solution for this problem is the usage of appropriate views. Views are stored queries that are executed at every access and thus form a virtual table. This is what the term *schema mapping* is used for. See for instance [Rahm and Bernstein, 2001].

Investigations of "cross-formalism views" (i.e., views that map one formalism onto another one) have been made. As an example, see for instance [Shanmugasundaram et al., 2001] who discuss the creation of XML views for relational databases. [Alon et al., 2003] explore certain problems that can arise during this process.

Logical Consistency

We come back to contradictions in logical sense. From now on we will assume that belief bases are syntactically compatible. What remains is to actually resolve logical conflicts. This is what some authors call *data integration* [Naumann and Häussler, 2002].

Under the term *logical conflicts* we understand conditions that either lead to falsity:

$$\bigcup_i KB_i \models \bot$$

or at least the violation of user-defined *side constraints* C:

$$\bigcup_i KB_i \not\models C$$

To understand the second case, imagine two relational tables, each with a column that stores unique values, i.e., there must not exist two distinct tuples that have the same value for this field. Then each table alone can be perfectly consistent whereas when merged, duplicates arise.

This is not only true for unique key constraints. Check constraints that involve a sum or average computation can be satisfied for two disjoint tables but violated when merged. This is illustrated with an example.

Example 4.2. Imagine two companies with typical customer databases. One of their constraints is that customers must not owe too much money. More precisely, there is some limit L such that the sum of all outstanding accounts must not exceed this value. Assume that the constraint is satisfied in both databases. However, if they are merged due to a company fusion, the constraint can be violated.

Take a look at the following tables. (a) and (b) are the source tables that are merged in (c). Assume a limit of $500,00$ Euros in both companies as well as in the merged one.

(a)

customer_no	sum(accounts)
0000014	480,00
0000125	195,00
0000402	250,00

(b)

customer_no	sum(accounts)
0000224	170,00
0000380	380,00
0000402	370,00

(c)

customer_no	sum(accounts)
0000014	480,00
0000125	195,00
0000224	170,00
0000380	380,00
0000402	620,00

Obviously, the constraint is satisfied in (a) and (b) but violated in (c) by customer 0000402 (last tuple).

In [Alchourròn et al., 1985], this type of contradiction is regarded in the following way. The side constraints are just represented as satisfiable set of formulas C. Then the result of a merging operator is a belief base that is not only consistent, but that also satisfies these constraints. Formally:

$$\circ^n(K) \models C$$

We will see later on how this model-theoretic definition can be adjusted for belief bases in form of answer sets (see Section 4.5).

Data Cleanness

While logical inconsistency in our sense leads to constraint violation that can automatically be detected (though not resolved), there may exist data sources that satisfy constraints which are either not explicitly encoded due to a lack in the quality of the data model or which cannot even be encoded.

Clearly, a good data schema should be complete in the sense that all important constraints are actually implemented. However, since we use third-party sources and experience has shown that some of them lack such a complete design, we also need to deal with this kind of contradictions.

An example for a constraint that cannot be explicitly implemented but which is the result of good practice while using the database is given below.

Example 4.3. Imagine relational datasets that incorporate address fields. It is a typical problem that the same address is written differently in several tuples. For instance, one abbreviates "street" and the other one does not. A common solution is to outsource addresses into a separate address table and refer to its entries whenever an address field is needed.

But this works only as long as we deal with a single database. If several sources are merged, the same problem arises again since each source will come with its own address table and thus it is possible that during merging, different versions find its way into the merged address repository.

The process of *data cleansing* deals with the removal or unification of different entries that refer to the same real-world object [Gravano et al., 2003]. The detection of such artefacts is tough since no constraints are violated in strict technical sense. It is rather one that requires considerations on a semantic level and constraint satisfaction is result of good practice in database usage. Hence, automatic procedures need to implement domain knowledge to a certain degree.

Comparison of the Types of Incompatibility

The distinction between the types of incompatibility is essential for what follows. The first problem, syntactic incompatibility between the sources, is *not* a consequence of the merging process but existed prior. That is, the belief bases per se are incompatible due to their formats. In the second and third case however, inconsistencies are actually consequences of the merging process.

To illustrate this, consider the following example (look at tables (a) and (b) only for now).

Example 4.4. Assume we have two tables that store persons together with attributes of their physical appearance, for instance their body height.

(a)

Name	Height
Homer	1,82m
Marge	1,78m
Bart	1,67m

(b)

Name	Height
Homer	5,97ft
Marge	5,84ft
Bart	5,40ft

(b')

Name	Height
Homer	1,82m
Marge	1,78m
Bart	1,65m

Source tables (a) and (b) are syntactically incompatible since the first one stores the height in meters and the latter one in feet. Note that this problem exists due to design of the tables. It is not consequence of the merging task.

Now let's assume we solve this problem by converting the height value of the second table from unit feet into meters. The result is depicted in table (b'). Now tables (a) and (b') are syntactically perfectly compatible. However, if they are merged using a simple union operation, we see that we have two different values for the body size of Bart. This inconsistency can be the consequence of human errors (someone has entered the wrong number), rounding errors or whatever. The point is that this conflict arises during merging of the tables.

4.4 Approach

In the last section we came to the conclusion that we have basically two types of inconsistencies. The first one is consequence of the fact that different belief bases are possibly incompatible in the sense that their encodings diverge. Second, conflicts can arise when information from two different sources contradict each other (where this kind was subdivided into logical inconsistencies and data cleanness).

The distinction is of special importance since the approaches for conflict resolution are very different. In the case of syntactic incompatibility it is sufficient to convert all the belief bases into a common format. This can (and should) be done independently for each source. In the illustrating example of the last section this was done by converting the height value of table (b) from feet into meters.

In the second case, conflict resolution must be tightly coupled with the merging task in the narrower sense since the inconsistencies are invisible in the source tables if treated independently. For our illustrating example, this means, when the tables are combined, we must decide if Bart is 1,65m or 1,67m. There are several possible strategies. A very simple one could be to trust one of the sources and discard the information from the other one. A more advanced strategy could be to take the average of the values, i.e., Bart is 1,66m in the merged table. The selection of an appropriate strategy is strongly application dependent.

Even though the details are still open, we have seen that we need basically two mechanisms. First, we need a mapping of the individual sources onto some kind of *common representation formalism*, second, we need a conflict resolution strategy during merging.

4.5 Task Definition

At this point we can define the belief merging task. For that purpose we formalize the intuitive description from Section 4.4.

First we introduce the notation that will be used below. The single items are described in more detail in the next subsections.

Notation. We will use the following notation for describing the elements of the merging task. Our belief bases are given as vector of logic programs $\pi = (P_1, \ldots, P_n)$. The common signature $\Sigma^C = (\Sigma_c^C, \Sigma_p^C)$ consists of a set of constant and a set of predicate symbols. The mappings are given as vector of functions $\mu = (\mu_1, \ldots, \mu_n)$ of the same dimensionality as π. Finally, $\omega = (\circ_1, \ldots, \circ_m)$ is our vector of merging operators.

In the general part of this chapter we followed the model-theoretic approach of many articles like [Alchourròn et al., 1985]. Now we change our point of view with respect to the upcoming implementation. From now on we assume that the belief bases are given as logic programs P_i. Their answer sets are regarded as *belief sets*, i.e., as the knowledge stored within the programs. Clearly, due to dlvhex' support for external atoms, this enables the user to write a wrapper for arbitrary types of knowledge bases (e.g., relational databases, XML files, text files) - for more details see Section 2.3.

To stress this point again, we are *not* going to merge programs (this would be a syntactic approach), but we merge the answer sets of the programs, i.e., we work on the *semantic level*.

In order to simplify the subsequent definitions, we introduce the following shortcut.

Definition 4.3. Let Σ be a signature. Then the answer repertoire $\mathcal{A}(\Sigma)$ over this signature is:

$$\mathcal{A}(\Sigma) = 2^{Lit_\Sigma}$$

The answer repertoire is the set where answer sets over this signature come from. That means, not all elements from the repertoire are actually answer sets, but each answer set is in $\mathcal{A}(\Sigma)$, i.e.,:

$$AS(P) \subseteq \mathcal{A}(\Sigma^P) \text{ for a program } P.$$

This is simply the power set of the set of literals over Σ, but this abbreviation will make subsequent formulas much more readable.

Mappings

In terms of HEX programs, each P_i may use a proprietary signature of certain predicate names and constants. This is denoted as:

$$\Sigma^{P_i} = \langle \Sigma_p^{P_i}, \Sigma_c^{P_i} \rangle$$

Now we have to fix a so called *common signature*, denoted as:

$$\Sigma^C = \langle \Sigma_p^C, \Sigma_c^C \rangle$$

This is a set of predicate names and constants that is expressive enough to represent the information stored in *any* of the bases. Then, all we have to do is to *map* the single sources onto the common signature. An essential side condition is that everything that is semantically equivalent needs to be syntactically equivalent and the other way round. Let's denote the abstract semantics of an answer set or set of answer sets X as $\tilde{X}$. Formally, such mappings μ_i are defined as functions.

Definition 4.4. A mapping is a function

$$\mu_i : 2^{\mathcal{A}(\Sigma^{P_i})} \to 2^{\mathcal{A}(\Sigma^C)} \; s.t. \; 2^{\mathcal{A}(\tilde{\Sigma}^{P_i})} = 2^{\mathcal{A}(\tilde{\Sigma}^C)}$$

Thus, we map sets of answer sets over the individual signatures Σ^{P_i} to sets of answer sets over the common signature Σ^C.

Note that in fact the mapping definition could be weakened. Definition 4.4 requires μ_i to map *any* subsets of the set of all literals onto the common signature. However, in practice some combinations will never occur due to internal side constraints. Thus it would be completely sufficient to define:

$$\mu_i' : \{AS(P_i)\} \to 2^{\mathcal{A}(\Sigma^C)}$$

That means, not *arbitrary* sets are mapped, but only those that are actually answer sets.

This is illustrated with an example.

Example 4.5. The program

$$\begin{aligned} P = \{ & friend(john). \\ & friend(joe). \\ & friend(jack). \\ invite(X) \vee discard(X) \; & \leftarrow \; friend(X). \\ & \leftarrow \; select(X1), select(X2), select(X3), \\ & \quad X1 \neq X2, X1 \neq X3, X2 \neq X3. \} \end{aligned}$$

is again motivated by a similar (but advanced) program in [Eiter et al., 2005]. It will randomly select one or two out of three friends and invites them. The first mapping definition require the mapping to be defined for

$$\{invite(john), invite(joe), invite(jack)\}$$

However, this set of literals will never occur as an answer set in practice because it invites 3 persons.

Even though the definition could be weakened in theory, we will stick with the initial suggestion since in practice, the restriction to actual answer sets will probably not simplify the mapping rules. This was discovered during the experiments made in Section 6 and comes from the fact that most predicates can be mapped fairly independently from others.

Merging of Belief Bases using Operators

Now we can safely assume that all P_i deliver answer sets over the same signature Σ^C, and that semantically equivalent information is also syntactically equivalent. If this is not the case we can simply apply μ_i in the following way: $\mu_i(AS(P_i))$.

The next step is to actually merge the information from several sources. Consider two programs P_i and P_j. Each of them yields a set of answer sets $AS(P_i)$ resp. $AS(P_j)$. For the purpose of merging we introduce the concept of *operators*. These are again functions that map two sets of answer sets (each over the common signature Σ^C) to another set of answer sets. Formally this is defined as follows:

Definition 4.5. A binary operator is a function

$$\circ^2 : 2^{\mathcal{A}(\Sigma^C)} \times 2^{\mathcal{A}(\Sigma^C)} \to 2^{\mathcal{A}(\Sigma^C)}$$

This is somehow related to merging operators as defined in the literature, for instance in [Alchourròn et al., 1985]. But in contrast to that, our definition is answer set-based rather than model-based. This is already a preparation step for later sections. However, the basic definition of functions that map n inputs to one element of the same domain is unchanged.

The concept of binary operators can easily be generalized to an n-ary version:

Definition 4.6. An n-ary operator is a function

$$\circ^n : \left(2^{\mathcal{A}(\Sigma^C)}\right)^n \to 2^{\mathcal{A}(\Sigma^C)}$$

Note that this definition is the most general one that is possible. Since it maps *answers* (i.e., whole *sets of answer sets*) to new answers it can actually do anything that is computable. Other definitions, like the mapping of answer sets onto answer sets

$$\circ^n : (\mathcal{A}(\Sigma^C))^n \to \mathcal{A}(\Sigma^C)$$

are unnecessarily restrictive and were therefore discarded.

The definition could again be weakened. Instead of $\left(2^{\mathcal{A}(\Sigma^C)}\right)^n$ we could also use the subset $\bigtimes\{\mu_i(AS(P_i))\}$ as domain of an n-ary operator since it is not necessary to map *any* set of subsets of all literals to new answer sets, but it is sufficient to map those sets that will actually occur as answer sets. Others will never be passed as input to the operator. However, as discussed in the previous section, this simplification cannot be exploited in practice. Thus we stick with larger but easier to define domain.

Examples

In this subsection we present a few operator definitions exemplary in oder to enhance the intuitive understanding. We start with a very simple operator that does nothing else than computing the pairwise union of answer sets.

Example 4.6. Consider operator $\circ^2_\cup$ with the following definition:

$$\circ^2_\cup : 2^{\mathcal{A}(\Sigma^C)} \times 2^{\mathcal{A}(\Sigma^C)} \to 2^{\mathcal{A}(\Sigma^C)}$$

$$\circ^2_\cup(SAS_1, SAS_2) = \{AS_1 \cup AS_2 \mid AS_1 \in SAS_1, AS_2 \in SAS_2, AS_1 \cup AS_2 \not\models \bot\}$$

Its input parameters are two sets of answer sets over the common signature Σ^C called SAS_1 and SAS_2 respectively. For each pair of answer sets AS_1 and AS_2, such that $AS_1 \in SAS_1$ and $AS_2 \in SAS_2$, the union $AS_1 \cup AS_2$ is an answer set of the operator's result if it is logically consistent.

To illustrate how this operator actually works, consider two programs P_1 and P_2 with $AS(P_1) = \{\{p, q\}, \{p\}\}$ and $AS(P_2) = \{\{r\}, \{s\}\}$. Then we have that:

$$\circ^2_\cup(AS(P_1), AS(P_2)) = \{\{p, q, r\}, \{p, q, s\}, \{p, r\}, \{p, s\}\}$$

Even though the union operator is intuitively a first class operator since it keeps all the information of the input belief bases, it is practically difficult to realize. It is clear that the above definition can easily introduce violations of user-defined integrity constraints if the information sources to be merged are differing. This was demonstrated by the example in Section 4.3. A further problem is that inconsistent pairs of answer sets are completely discarded even though parts of them could be useful.

A well-known approach comes from the field of belief revision. The AGM postulates were introduced in [Alchourròn et al., 1985] and were since then often cited as the goal of belief revision. That is, an existing belief base shall be extended with a certain statement (a logic formula), such that the new formula holds and the existing knowledge is modified as little as possible. If the new information is consistent with the knowledge base it can be simply added. Otherwise modifications need to be done in order to avoid contradictions, since an inconsistent knowledge base would imply anything in classical logic and thus no useful information could be deduced anymore.

One of the subsequently developed belief revision operators is described in [Ginsberg, 1986]. Let ϕ be the new statement to be incorporated into the knowledge base KB. Then the Ginsberg operator takes the subset-maximal sub-theory of KB that is consistent with ϕ, i.e.,:

$$max_\subseteq\{KB' \subseteq KB \mid KB' \not\models \neg\phi\}$$

Since the result is in general not unique, the set of possible revised knowledge bases is considered as a disjunction. The WIDTIO approach (*when in doubt, throw it out*) is more restrictive in the sense that the cut of all possible worlds is computed rather than the disjunction. Other operators like the one defined in [Winslett, 1988] are rather model- than formula-based. We will give an example for an operator with a similar intention.

Example 4.7. The subsequently defined operator $\circ_{max_\subseteq}$ is similar to Ginsberg's or the WIDTIO approach.

$$\circ^2_{max_\subseteq} : 2^{\mathcal{A}(\Sigma^C)} \times 2^{\mathcal{A}(\Sigma^C)} \to 2^{\mathcal{A}(\Sigma^C)}$$

$$\circ^2_{max_\subseteq}(SAS_1, SAS_2) = max_\subseteq\{AS \mid AS_1 \in SAS_1, AS_2 \in SAS_2, AS \subseteq (AS_1 \cup AS_2), AS \not\models \bot\}$$

The result of $\circ_{max_\subseteq}$ contains all subset-maximal subsets of pairwise unions of elements $\in SAS_1$ and $\in SAS_2$ that are logically consistent.

Consider for instance:

$$SAS_1 = \{\{p, q\}, \{\neg p\}\}$$

and:

$$SAS_2 = \{\{\neg q\}, \{p\}\}$$

Then

$$\begin{aligned}\circ^2_{max_\subseteq}(SAS_1, SAS_2) &= max_\subseteq\{S \mid AS \in U \wedge A \subseteq AS \wedge S \not\models \bot\} = \\ &\quad \textit{with } U = \{\{p, q, \neg q\}, \{p, q\}, \{\neg p, \neg q\}, \{\neg p, p\}\} \\ &= \{\{p, q\}, \{p, \neg q\}, \{p, q\}, \{\neg p, \neg q\}, \{\neg p\}, \{p\}\} = \\ &= \{\{p, q\}, \{p, \neg q\}, \{\neg p, \neg q\}, \{\neg p\}, \{p\}\}\end{aligned}$$

Constraints

We come back to logical consistency regarding application specific side constraints as introduced in Section 4.3. If a relational table with two columns $tab(X, Y)$ which implements a unique key constraint for argument X (for instance a primary key constraint), the answer set

$$\{tab(1, a), tab(1, b)\}$$

is *not* inconsistent in logical sense since it does not contain complementary literals. Nevertheless the integrity constraint is violated.

However, how can the operator know about this kind of undesired conditions, if it does not lead to logical contradictions? This requires a minor modification of the operator $\circ_{max_\subseteq}$.

Example 4.8. The following modified version of the operator $\circ_{max_\subseteq}$ can deal with user-specified side constraints.

$$\circ^2_{max'_\subseteq} : 2^{\mathcal{A}(\Sigma^C)} \times 2^{\mathcal{A}(\Sigma^C)} \times 2^{\mathcal{A}(\Sigma^C)} \to 2^{\mathcal{A}(\Sigma^C)}$$

$$\circ^2_{max'_\subseteq}(SAS_1, SAS_2, \mathcal{C}) = max_\subseteq\{AS \mid AS_1 \in SAS_1, AS_2 \in SAS_2, AS \subseteq (AS_1 \cup AS_2), AS \not\models \bot, AS \in \mathcal{C}\}$$

It is again a binary operator. Observe that its signature does not correspond to our general operator Definition 4.6 since it depends on an additional parameter $\mathcal{C}$ of type $2^{\mathcal{A}(\Sigma^C)}$. This parameter is nothing more than a mathematical formalization of constraints. $\mathcal{C} \subseteq 2^{\mathcal{A}(\Sigma^C)}$ is the set of all answer sets that satisfy these constraints. If the union of two answer sets is not contained in $\mathcal{C}$, it is an example for a constraint violation as introduced in Section 4.3.

As an example, assume that predicate $p(X)$ must not be simultaneously *true* for X and Y with $Y \neq X$, i.e., p is subject to a unique constraint. Then

$$\mathcal{C} = \{AS \mid AS \in \mathcal{A}(\Sigma^C) \wedge \neg(\exists X, Y : X \neq Y \wedge p(X), p(Y) \in AS)\}$$

The operator's result contains all subsets of pairwise unions of elements $\in SAS_1$ and $\in SAS_2$ that are not only logically consistent, but also with respect to certain certain domain dependent constraints.

If operator

$$\circ^2_{max'_\subseteq}(\cdot,\cdot)$$

is applied on the belief bases

$$SAS_1 = \{\{p(a), q(a)\}, \{q(c)\}\}$$

and

$$SAS_2 = \{\{p(b)\}\}$$

with the constraints

$$\mathcal{C} = \{as \mid as \in \mathcal{A}(\Sigma^C) \wedge \neg(\exists X, Y : X \neq Y \wedge p(X), p(Y) \in as)\}$$

(i.e., there is a unique constraint for predicate p) then

$$\circ^2_{max'_\subseteq}(SAS_1, SAS_2) = \{\{p(a), q(a)\}, \{p(b), q(a)\}, \{q(c), p(b)\}\}$$

The answer sets $\{p(a), q(a)\}$ and $\{p(b), q(a)\}$ come from the union of $\{p(a), q(a)\}$ and $\{p(b)\}$ where either $p(a)$ or $p(b)$ needs to be thrown out due to the unique constraint.

Note that in the last example, operator $\circ^2_{max'_\subseteq}$ was considered to be binary even though it takes 3 arguments. However, it actually merges only two belief bases. The third parameter is a set of *constraints* rather than a belief base. This leads to the concept of *additional operator arguments*.

Operator Arguments

The example of $\circ^2_{max'_\subseteq}$ illustrated that some operators are not only mappings of belief bases (given as answers of programs) to new answers, but require additional information. In the previous examples this were constraints, but the concept can easily be generalized. This leads to our final definition of operators as they will be implemented in Chapter 5.

Definition 4.7. An n-ary operator with m additional arguments of types $\mathcal{D}_i$ $(0 \leq i < m)$ is a function

$$\circ^{n,m} : \underbrace{\left(2^{\mathcal{A}(\Sigma^C)}\right)^n}_{belief\ \ bases} \times \underbrace{\mathcal{D}_0 \times \ldots \times \mathcal{D}_m}_{additional\ \ parameters} \rightarrow 2^{\mathcal{A}(\Sigma^C)}$$

In Example 4.8, we have $m = 1$ with $\mathcal{D}_0 = 2^{\mathcal{A}(\Sigma^C)}$. Note that the type of the first (and only) additional parameter is equivalent to the types of belief base results (namely sets of answer sets). Nevertheless the two groups of parameters are conceptually different. Additional parameters can in general be of arbitrary types. This will be an important aspect for our implementation.

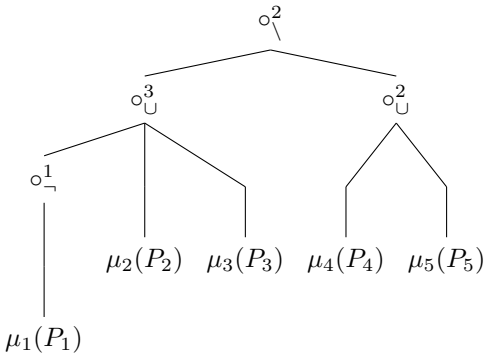
Merging Plans

In the last subsection we introduced the concept of operators that allows us to merge two or more belief bases. Now we come to an extension of this idea. *Merging plans* are strategies that allow merging of arbitrary many belief bases using a set of operators in a predefined order.

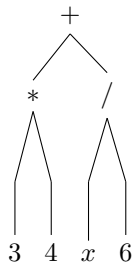
According to the definitions in Section 4.5, ω is a set of operators. However, this definition does not state how these operators are actually applied to the set of belief bases π. It is not clear at this point which operators get which belief bases as parameters.

After a set of two or more belief bases were merged, the result can again be treated as belief base and be the argument for a further operator application. Thus, merging plans need to be hierarchically organized. Formally, we can define them as *acyclic graphs*, where the leaf nodes are belief bases and the inner nodes are merging operators. The following example illustrates this.

Example 4.9. The following merging plan computes the negation of P_1 using the unary operator $\circ^1_\neg$. Then it computes both the union of P_1, P_2 and P_3 as well as the union of P_4 and P_5 (assume that there exists a binary and a ternary version of the union operator). Finally it subtracts the result of $\circ^2_\cup$ from the result of the ternary operator with the semantics of set difference.



This kind of nesting operators works in the same way as arithmetical expressions are organized. Compare the previous example with a typical syntax tree for terms in an arbitrary programming language:



The flow of information in the framework is shown schematically in Figure 4.1, where the number of operators, the depth of the merging tree and belief bases can vary of course.
Formally we can define merging plans as follows.

Definition 4.8. The set of merging plans $\mathcal{R}_{\pi,\omega}$ over a set of belief bases π and a set of operators ω is defined inductively as follows.

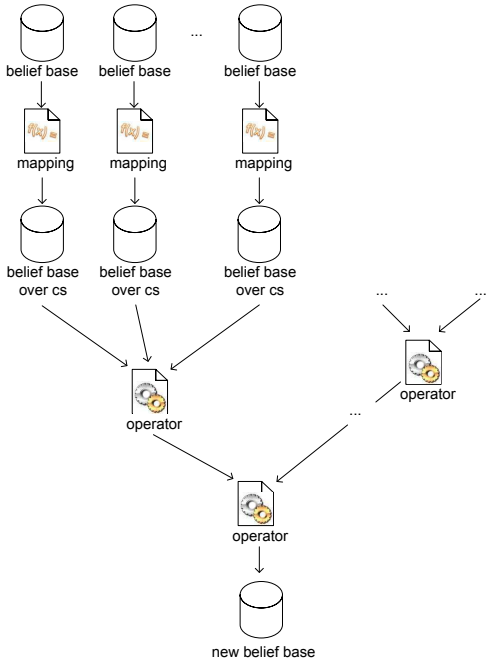


Figure 4.1: Flow of information in the belief merging framework

(i) Each atomic merging plan $R \in \pi$ is a merging plan, i.e., $\pi \subseteq \mathcal{R}_{\pi,\omega}$

(ii) If $\circ_i^{n,m} \in \omega, s_j \in \mathcal{R}_{\pi,\omega}$ and $a_k \in \mathcal{D}_i$ for $1 \leq j \leq n, 1 \leq k \leq m$, then

$$(\circ_i^{n,m}, s_1, \ldots, s_n, a_1, \ldots, a_m) \in \mathcal{R}_{\pi,\omega}$$

(iii) Merging plans are *only* constructed according to rules (i) and (ii)

If $\mathcal{R}^k_{\pi,\omega}$ denotes the set of all merging plans that can be built with up to k nested operator applications, then $\mathcal{R}^0_{\pi,\omega}$ obviously is just the set of belief bases. $\mathcal{R}^1_{\pi,\omega}$ is the same set plus all tuples of kind

$$(\circ_i^{n,m}, s_1, \ldots, s_n, a_1, \ldots, a_m)$$

where $\circ_i^{n,m} \in \omega, s_i \in \pi, a_i \in \mathcal{D}_i$. That is, we apply an n-ary operator $\circ_i^{n,m}$ (with m additional parameters) on n belief bases and m additional arguments of the according domains. Clearly all merging plans in this set use either 0 or 1 nested operator applications.

In general, a merging plan in $\mathcal{R}^k_{\pi,\omega}$ is an operator that is applied on merging plans in $\mathcal{R}^{k-1}_{\pi,\omega}$. Since all those merging plans are formed with up to $k-1$ nested operators, the merging plans in $\mathcal{R}^k_{\pi,\omega}$ use up to k nestings, which can easily be proved by induction. Clearly, $\mathcal{R}_{\pi,\omega} = \bigcup_{k \geq 0} \mathcal{R}^k_{\pi,\omega}$.

The formal version of the merging plan in Example 5.6 is therefore:

$$R = (\circ^2_{\setminus},\ (\circ^3_{\cup},\ (\circ^1_{\neg},\ P_1),\ P_2,\ P_3), (\circ^2_{\cup},\ P_4,\ P_5))$$

Merging Task

This leads to the final definition of the merging task. It is straightforward to put all the formalizations from the previous subsections together.

Definition 4.9. A merging task is a quintuple

$$\langle \pi, \Sigma^C, \mu, \omega, R \rangle$$

where π are the belief bases, Σ^C is the common signature, μ are the mappings of π onto Σ^C, ω is a set of operators and $R \in \mathcal{R}_{\pi,\omega}$ is a merging plan over π and ω.

What remains to define is the formal semantics of a merging task.

Definition 4.10. The semantics of the merging task, denoted as $\| \cdot \|$, is defined as follows:

(i) $\| \langle \pi, \Sigma^C, \mu, \omega, R \rangle \| = [\mu_i(AS(R))]_{\Sigma_p^C}$ iff $R \in \pi$

(ii) $\| \langle \pi, \Sigma^C, \mu, \omega, R \rangle \| = [\circ_i^{n,m}(\| \langle \pi, \Sigma^C, \mu, \omega, s_1 \rangle \|, \ldots, \| \langle \pi, \Sigma^C, \mu, \omega, s_n \rangle \|, a_1, \ldots, a_m)]_{\Sigma_p^C}$
otherwise, i.e., $R = (\circ_i^{n,m}, s_1, \ldots, s_n, a_1, \ldots, a_m)$

where $[SAS]_{\Sigma_p^C}$ denotes the projection of SAS onto atoms over Σ_p^C, i.e.,

$$[SAS]_{\Sigma_p^C} := \{\{p(a_1, \ldots, a_n) \in AS \mid (p = p' \vee p = \neg p') \wedge p' \in \Sigma_p^C\} \mid AS \in SAS\}$$

Clearly, the result of a merging task can be considered to be a new belief base π^* by constructing an appropriate formula, i.e.:

$$\pi^* = form(\| \langle \pi, \Sigma^C, \mu, \omega, R \rangle \|)$$

Informally, if R is an atomic merging plan (i.e., a belief base), then it can be evaluated directly and its semantics is just the set of answer sets the according program yields, but mapped to the common signature.

Otherwise, R contains at least one operator application. Then its semantics is defined as the result of the topmost operator, applied on the results of the sub-merging plans.

As a final example, let us reconsider the formal merging plan from the previous section.

Example 4.10. Let R be the merging plan:

$$R = (\circ_{\setminus}^2,\ (\circ_{\cup}^3,\ (\circ_{\neg}^1,\ P_1),\ P_2,\ P_3), (\circ_{\cup}^2,\ P_4,\ P_5))$$

and let

$$P_1 = \{a.,\ b.\}, P_2 = \{x.,\ y.\}, P_3 = \{a.,\ c.\}, P_4 = \{a.,\ x.\}, P_5 = \{c.,\ x.,\ y.\}$$

The complete input for the merging task is the 5-tuple

$$\langle \{P_1, \ldots, P_5\}, \Sigma^C, \mu_{id}, \omega, R \rangle$$

where the common signature contains the propositional (0-ary) atoms a, b, c, x and y, all mappings in μ are identity functions (since all belief bases use already the common vocabulary), and $\omega = \{\circ_{\cup}^3, \circ_{\setminus}^2\ \circ_{\neg}^1\}$.

Now we compute the semantics of this merging task as follows. For the sake of readability, we will omit π, Σ^C, μ and ω, and restrict ourselves to the merging plans, even though, strictly

speaking, we would have to pass the whole quintuple for each application of the semantics function denoted as $\| \cdot \|$.

$$
\begin{aligned}
& \| \left\langle \{P_1, \ldots, P_5, \Sigma^C, \mu_{id}, \omega, R\} \right\rangle \| = \\
= \quad & \circ^2_{\setminus}(\ \|(\circ^3_{\cup}, (\circ^1_{\neg},\ P_1),\ P_2,\ P_3))\|,\ \|(\circ^2_{\cup}, P_4, P_5)\|\ ) = \\
= \quad & \circ^2_{\setminus}(\ \circ^3_{\cup}(\|(\circ^1_{\neg},\ P_1)\|,\ \|P_2\|,\ \|P_3\|),\ \|(\circ^2_{\cup}, P_4, P_5)\|\ ) = \\
= \quad & \circ^2_{\setminus}(\ \circ^3_{\cup}(\|(\circ^1_{\neg},\ P_1)\|,\ \|P_2\|,\ \|P_3\|),\ \circ^2_{\cup}(\|P_4\|, \|P_5\|)\ ) = \\
= \quad & \circ^2_{\setminus}(\ \circ^3_{\cup}(\circ^1_{\neg}(\|P_1\|),\ \|P_2\|,\ \|P_3\|),\ \circ^2_{\cup}(\|P_4\|, \|P_5\|)\ ) = \\
= \quad & \circ^2_{\setminus}(\ \circ^3_{\cup}(\circ^1_{\neg}(\|P_1\|),\ \|P_2\|,\ \|P_3\|),\ \circ^2_{\cup}(\|P_4\|, \|P_5\|)\ ) = \\
= \quad & \circ^2_{\setminus}(\ \circ^3_{\cup}(\circ^1_{\neg}(\|P_1\|),\ \|P_2\|,\ \|P_3\|),\ \circ^2_{\cup}(\{\{a, x\}\}, \{\{c, x, y\}\})\ ) = \\
= \quad & \circ^2_{\setminus}(\ \circ^3_{\cup}(\circ^1_{\neg}(\{\{a, b\}\}),\ \{\{x, y\}\},\ \{\{a, c\}\}),\ \circ^2_{\cup}(\{\{a, x\}\},\ \{\{c, x, y\}\})\ ) = \\
= \quad & \circ^2_{\setminus}(\ \circ^3_{\cup}(\{\{\neg a, \neg b\}\},\ \{\{x, y\}\},\ \{\{a, c\}\}),\ \circ^2_{\cup}(\{\{a, x\}\},\ \{\{c, x, y\}\})\ ) = \\
= \quad & \circ^2_{\setminus}(\{\{a, \neg a, \neg b, c, x, y\}\},\ \{\{a, c, x, y\}\}) = \\
= \quad & \{\{\neg a, \neg b\}\}
\end{aligned}
$$

4.6 Intention of the Framework

The proposed framework allows the user to merge any belief bases as long as suitable operators are provided. Its actual strength is the possibility to try out several merging plans very quickly. Therefore it is especially useful if there exist different merging strategies and it is unclear which of them is the best for the current task.

Without the framework it would be necessary to implement each merging variant from scratch. This includes many routine tasks like parsing of intermediate results and coordinating the information flow through the different operators.

Now it is possible to focus on the most interesting part, namely developing the merging strategies themselves. The technical details are hidden behind a user-friendly merging plan language which allows the specification of the information flow in a declarative manner.

After the user has finished the experiments and knows which of the merging plans is suited for the applications, he can still decide to directly uses the output of the framework, or if the winning merging plan is reimplemented by hand due to performance reasons.

To summarize, the intention of the framework is to give the user the possibility to make experiments with different merging strategies without dealing with technical details that are not in direct connection to the actual merging strategies.

This concludes our formal definitions and leads to the implementation in dlvhex.

In theory, there is no difference
between theory and practice.
But, in practice, there is.
Jan L. A. van de Snepscheut

Chapter 5

Implementation

In the last chapter we defined the belief merging framework formally. To summarize, the input of the merging task is given as 5-tuple

$$\langle \pi, \Sigma^C, \mu, \omega, R \rangle$$

where π is a vector of input belief bases, Σ^C the common signature, μ the vector of mappings of the individual belief bases to the common signature, ω a vector of merging operators to be used and R is the merging plan.

In this chapter we will show how this task definition can actually be used to implement a corresponding framework for dlvhex.

The individual belief bases $\pi = (P_1, \ldots, P_n)$ are given as HEX programs. Note that even though this appears as restriction at first glance, we can still easily use any knowledge base as input. For instance relational databases, XML files, object-orientated databases, ontologies written in description logics or any other source, as long as it is written in a knowledge representation formalism that is accessible from dlvhex through a suitable plugin.

Thus we can safely assume that information sources are always given in form of sets of answer sets.

5.1 Architectural Overview

In this section we introduce the architecture of the implementation. We use two reference figures. Figure 5.1 shows the plugin as part of the workflow from user's perspective whereas Figure 5.2 shows the internal structure. The internals of the mergingplugin were already shown in Figure 3.2. Now we also explain the part for operator application and the mpcompiler, that are also part of the plugin and were skipped in Chapter 3.

The top of Figure 5.1 shows the input for the merging task consisting of five elements, as described in the general part of this chapter. The most basic input that is mostly provided by third parties are the belief bases in the narrower sense. They are mapped to the (unique) common signature using zero to many mappings (or two to many, for reasonable merging tasks).

A further input is the set of operators, which are given by some library methods that compute certain functions. Last but not least, the merging plan itself is the glue that binds all

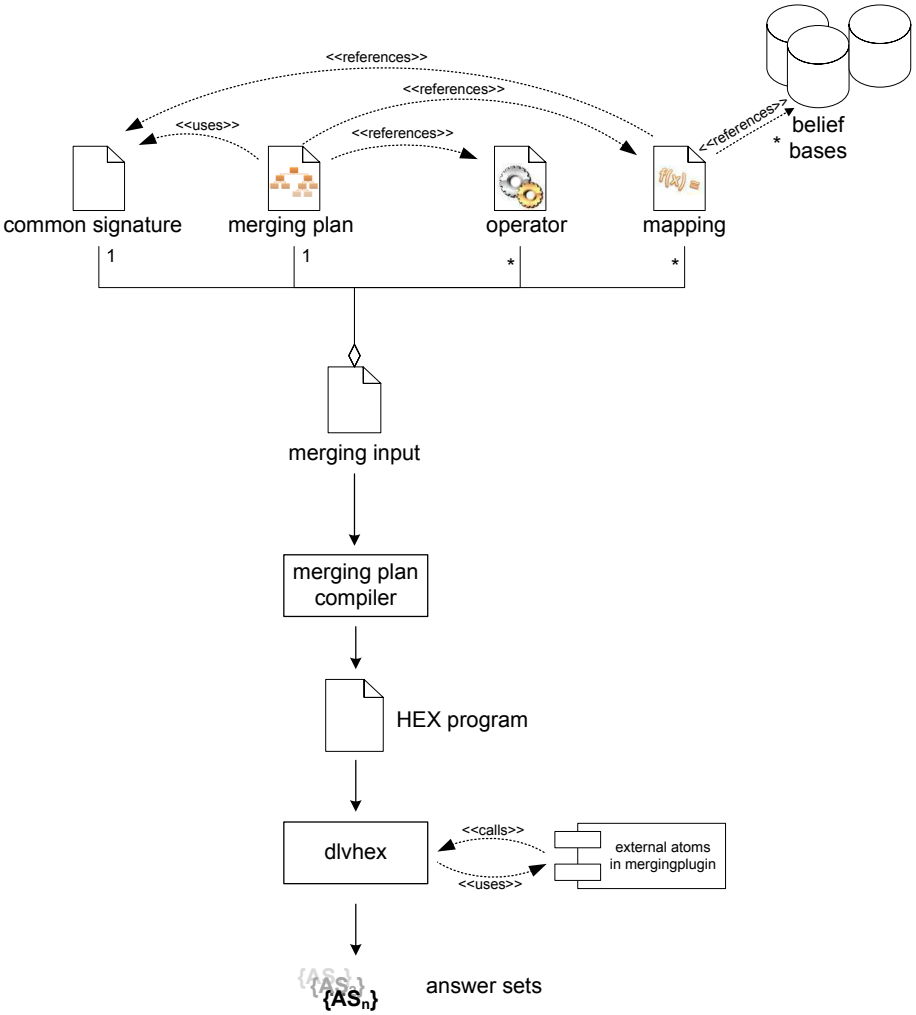


Figure 5.1: Architectural overview from user perspective

the other components together. It references the operators and the belief bases (through their corresponding mappings) and describes how they need to be combined.

The five input components together form the merging input that is visualized as symbolic file. However, in fact it is not a single file as we will see in later sections, but for now this point of view is sufficient.

Now the merging input can be passed to the *merging plan compiler*, which is a component that was not mentioned so far. As we will see, the 5-tuple given above is not suited for being directly processed by dlvhex. Intuitively this is clear since not all of the input components are sets of rules. Thus the merging plan compiler interprets and translates it into an appropriate format that is basically a HEX program, though it is a very complex program that makes excessive use of certain external atoms. Especially the atoms concerning nested HEX programs (see Chapter 3) are fundamental in this step, that are implemented in the so called mergingplugin. As the figure suggests, dlvhex depends on them in order to compute the result.

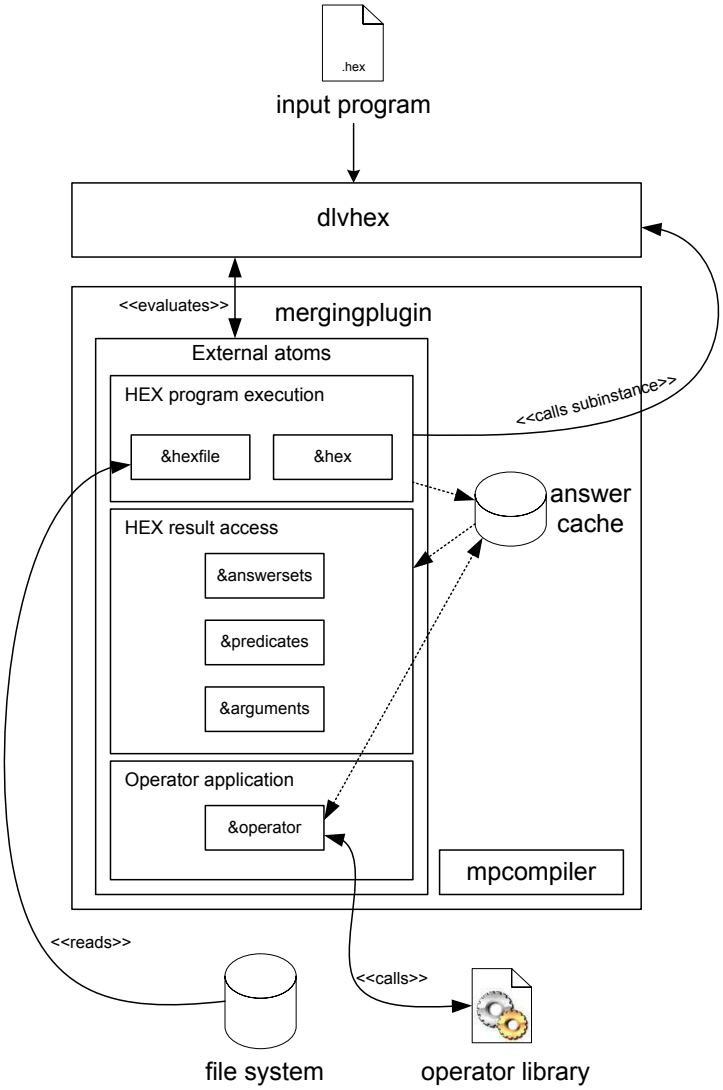


Figure 5.2: mergingplugin internals with &*operator* and mpcompiler

The remaining part of this chapter is organized as follows. First we describe how the five input components will actually be implemented. Then the translation of this input into a semantically equivalent HEX program is explained, which is the basis of the merging plan compiler.

5.2 Operator Implementation

We first discuss the implementation of an operator of kind

$$\circ^n(s_1, \ldots, s_n, p_1, \ldots, p_m)$$

since this is fairly independent from the other parts. Recall the definition from Section 4.5. An n-ary operators maps n sets of answer sets AS_i and possibly up to m additional arguments of

any type onto a new set of answer sets. Thus our implementation needs to provide these features.

In Section 3.5 we have already discussed how the mechanism of nested HEX programs can be implemented as a dlvhex plugin. We introduced the concept of a *handle*, that refers to the components of HEX program results. This enabled us to access answers of HEX programs, answer sets within answers, predicate names and finally also the arguments of predicate occurrences.

An External Atom for Operator Applications

Now we extend this functionality by providing an additional external atom for operator calls. In total this atom needs to take the following parameters: an operator name, a list of answers that are passed to the operator (each represented by a handle) and possibly additional parameters of any type.

Since knowledge bases are assumed to be sets of answer sets and they are represented by integer handles, we can reduce the theoretical definition of an n-ary operator $\circ_i^n$ to the following external atom:

$$\&operator_{\circ_i^n} : \mathbb{N}^n \to \mathbb{N},$$

The operator takes n handles and returns another handle to the operator's result. Then we can use the atoms introduced in Chapter 3 to look into the answer and continue computation.

As we will see in Section 5.2, it is straightforward to generalize this definition with respect to the operator's arity. Sometimes it is reasonable to provide operators of variable arity. For instance in Section 4.5 we used both a binary and a ternary union operator. It would be painful to implement actually two different versions since this would require code copying with the usual troubles, e.g., updating each of them if a bug is fixed. Thus we modify the signature to:

$$\&operator_{\circ_i} : A \to \mathbb{N}.$$

where $A = \langle a_1, \ldots, a_n \rangle, a_i \in \mathbb{N}$ is the list of answers that is passed to the operator. Note that now the signature of $\&operator_{\circ_i}$ stays the same, even if a different number of answers is passed. In other words, n is variable.

So far we introduced an external atom for one specific operator $\circ_i$. It would be very inconvenient for later sections to have a separate atom for each operator. Thus we modify the definition slightly to:

$$\&operator : String \times A \to \mathbb{N}.$$

This enables us to have just one atom for an arbitrary number of different operators. It just gets the operator's name as a second parameter.

Now we look at the additional parameters. These are somehow problematic since they can be of *any* type. We said that we want only one external atom for all operators. But how can that work if each operator can have a *different* number of parameters that are additionally of *arbitrary* types? This works by a simple assumption. In most cases, operators will get only very few parameters. And even though they can be of any type in theory, in practice they will mostly be numbers or logic formulas (constraints). Both can easily be represented as strings. However, we will not use a list of string parameters but follow an approach that is similar to the way how many computer applications and function libraries implement user specific configurations, namely as key-value pairs. This is a very expressive formalism that allows the user to encode any information quite easily. Compared with a fixed list of m string parameters, it additionally enables the user to have things like optional arguments. And due to the fact that merging input files look similar to typical configuration files, as we will see, this is a very natural extension.

In order to provide a definition that allows the user to pass an *arbitrary* number of key-value-pairs we outsource the parameters P as we did this in case of the answers A.

Our final definition of the predicate is therefore given as follows.

Definition 5.1. The external atom *&operators* maps an operator name, a list of n answers and m key-value pairs (over strings) onto a new answer, i.e.:

$$\&operator : String \times A \times P \to \mathbb{N}$$

with $A = \langle a_1, \ldots, a_n \rangle, a_i \in \mathbb{N}$ and $P = \{(k_1, v_1), \ldots, (k_m, v_m)\}, k_i, v_i \in String$

The first parameter is the name of the operator to call, the n integers are handles to the answers of prior HEX program executions and the m tuples over strings are key-value pairs for additional parameters to be passed. The atom returns another integer which is a handle to the operator's result.

Parameter Passing

Now we discuss how the parameters can be actually implemented and passed to *&operator*. The operator name is clearly a string, which can directly be written as constant within the HEX program. The list of answers is simply a list of integers since we represented them by handles to entries of the plugin's answer cache. dlvhex offers a very elegant way for passing *lists* of values, namely by using predicate names as input parameters. We just introduce a unique name for each list parameter and make the according predicate *true* for all values we want to pass. Actually this predicate needs to be binary. Since we do not pass *sets* but *lists*, we need the second parameter to store the order in which the values are passed. This is demonstrated by Example 5.1.

Example 5.1. Assume that the external atom *&exAt* is defined and expects one predicate as input parameter. Then the program:

$$\begin{aligned} P = \{ & op(0, 100). \\ & op(1, 200). \\ & op(2, 260). \\ & \ldots \leftarrow \&exAt[op](\ldots)\} \end{aligned}$$

passes the values 100, 200 and 260 *in this order* to the atom.

Observe that with this kind of parameter passing, the parameter number is clearly variable, which is a great advantage as we discussed above.

The usage of the *&operator* atom will now be illustrated with an example. Assume that an operator named *union* with the semantics demonstrated in Section 4.5 was defined. Consider Example 5.2.

Example 5.2. This program demonstrates the usage of *&operator*. Note that we changed the operator's name from $\circ^2_\cup$ in the last section to *union*. This should point out that we now talk about the implementation in dlvhex where we have to use string literals rather than mathematical symbols.

$$\begin{aligned} P = \{ handle1(H_1) &\leftarrow \&hex[\text{“}a.\ b.\text{”}, \text{“”}](H_1). \\ handle2(H_2) &\leftarrow \&hex[\text{“}c.\text{”}, \text{“”}](H_2). \\ opInput(0, H_1) &\leftarrow handle1(H_1). \\ opInput(1, H_2) &\leftarrow handle2(H_2). \\ handle3(H_3) &\leftarrow \&operator[\text{“}union\text{”}, opInput, additionalArgs](H_3). \\ result(Pred, Arity) &\leftarrow handle3(H_3), \&predicates[H_3](Pred, Arity).\} \end{aligned}$$

The first two lines of code execute two embedded HEX programs. The first one has the answer $\{\{a, b\}\}$ and the second one $\{\{c\}\}$. The following two lines define a binary predicate *opInput* such that 0 is paired with the handle to the result of program 1 and 1 with the result of program 2. In other words, the 0-th parameter to the operator is the result of the first program whereas the 1-st one is the answer of the second.

Finally the operator is actually applied. Note that the binary predicate *additionalArgs* is passed to the operator. This predicate is actually never defined. However, this does not matter since &*operator* expects its 3rd parameter to be a predicate defining all the key-value pairs that are needed by the operator. Because of the fact that the union operator does not expect (or interpret, if passed) additional parameters, it is irrelevant what we pass here.

The result is again a handle (called H_3 in the example) to the operator's return value, which is written into the answer cache. It is used in the last line of code to look into the result and extract all the predicates together with their arities. In this case, the final result will be:

$$\{\{result(a, 0), result(b, 0), result(c, 0)\}\}$$

(plus the intermediate atoms concerning parameters and handles; they have been filtered out), i.e., the answer consists of one answer set that contains the propositional atoms a, b and c.

Operator Plugins

We discussed how we can use an external atom to call operators. This atom is implemented in the same dlvhex plugin as the external atoms for nested HEX program execution that were introduced in Chapter 3. We called it mergingplugin. This is not only very convenient but even necessary, since the internals of the atom need to resolve the handles and access the answer sets behind, that are cached within this plugin.

However, what we ignored so far is the fact that the operators have to be defined somewhere. For instance, in the above example the operator named *union* is called and we silently assumed that it actually computes the set union. Now we discuss how we can implement operators.

The easiest way would definitely be to include it directly in the implementation of the &*operator* atom. We could read the first parameter of the atom, that is the operator's name, and execute the according functionality. Though, this solution is nasty when one wants to expand the framework with new operators.

Thus it is desirable to have the possibility for adding operators independently from the mergingplugin. In other words, we want to write *"plugins for the plugin"*, namely operator plugins for the mergingplugin.

This is exactly what was finally realized. The C++ framework `boost.org` provides functions for calling shared object libraries. dlvhex uses this mechanism for loading its plugins on startup. It expects plugin libraries to implement a method with the following signature:

```
extern "C" PluginInterface* PLUGINIMPORTFUNCTION()
```

dlvhex opens all libraries in its system and user plugin directories (plus custom directories that can be given as command-line arguments) and calls this method for each. This returns a pointer to an instance of the plugin class.

We use a similar mechanism for importing operators. When the mergingplugin starts up, it iterates through all libraries found in dlvhex' plugin directory and calls the following method for each (if present):

```
extern "C" std::vector<dlvhex::merging::IOperator*> OPERATORIMPORTFUNCTION()
```

The method returns a vector of pointers to operator implementations. The use of a *vector* enables the user to implement arbitrary many operators within one library.

Note that a library can simultaneously provide both import functions. That is, it can both implement a dlvhex plugin and operators for the mergingplugin. Figure 5.3 summarizes the architecture.

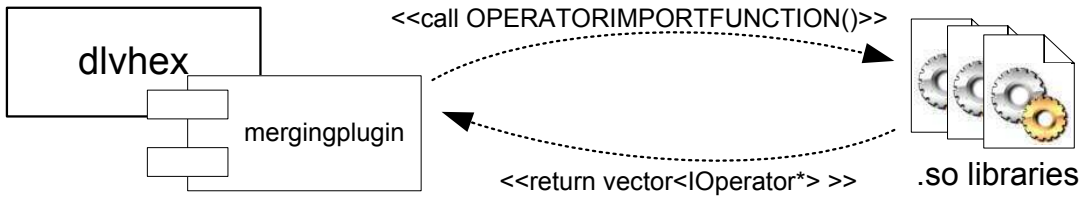


Figure 5.3: Operator plugins

The operator import function does already suggest that operators need to implement an interface called IOperator. This interface contains abstract versions of methods that an operator needs to provide. Listing 5.1 shows the definition of this class.

```
class IOperator{
public:
    class OperatorException{
    private:
        string msg;
    public:
        OperatorException(const string& m) : msg(m){}
        string getMessage(){ return msg; }
    };
    virtual string getName() = 0;
    virtual HexAnswer apply(int arity,
                            vector<HexAnswer*>& answers,
                            OperatorArguments& parameters) = 0;
};
```

Listing 5.1: interface IOperator

Thus, an operator needs to provide a name (string getName()) and an apply method for actual application. The arguments of the operator are its arity, a vector of HexAnswer, which is defined as vector of atom sets (just as defined in Section 4.5), and finally the OperatorArguments. This is a vector of key-value pairs for passing additional parameters. The arity will contain the actual number of answers that are passed to the operator. This allows the implementation to check if a unary, binary, ternary or - generally - an n-ary version is required and set its internal loop conditions appropriately. The result of the method is again a HexAnswer.

The implementation of an operator may throw an OperatorException if it cannot perform its task. This could be the case if required input parameters are missing, or if there is no operator version of the required arity. For instance, the implementation of a set difference operator will probably provide a binary version only.

At this point we have everything we need in order to execute the examples given in Section 5.2 as expected. We are able to load external operators from shared object libraries. By calling the method getName() we can determine their names. Thus, the mergingplugin can construct a list of (name, IOperator*) tuples on startup that can be used to call the correct operator within the implementation of &*operator*.

Expressiveness of our Operator Implementation

We now show that our implementation of operators corresponds to Definition 4.7 in Section 4.5.

Proposition 1. The external atom &*operator* allows the implementation and application of operators that exactly correspond to the operators definable with respect to Definition 4.7.

Proof. Completeness. We need to show that atom &*operator* allows the implementation and application of arbitrary operators of kind

$$\circ^{n,m} : \underbrace{\left(2^{\mathcal{A}(\Sigma^C)}\right)^n}_{belief\ \ bases} \times \underbrace{\mathcal{D}_0 \times \ldots \times \mathcal{D}_m}_{additional\ \ parameters} \rightarrow 2^{\mathcal{A}(\Sigma^C)}$$

Let *op* be such an operator with arity n and with m additional parameters $d_i \in \mathcal{D}_i$, $1 \leq i \leq m$. Further assume that this operator needs to be applied on (atomic or composed) belief bases $P_1, \ldots, P_n$. The remaining part of the completeness proof consists of two sub-proofs. We need to show that (i) *op* is definable within the framework and (ii) *op* is applicable on arbitrary arguments.

(i) We first show that our implementation allows the definition of this operator. We simply write a new C++ class which implements the interface IOperator (see Listing 5.1) and defines the methods getName and apply. As one can easily see in the interface definition, the apply method supports passing a vector of intermediate results of arbitrary length (and therefore especially of length n), as well as a data structure of type OperatorArguments, which is defined as set of key-value pairs over strings. But key-value pairs over strings allow the encoding of sequences of elements over arbitrary domains, since the key can be used as running index and the value contains a string representation of the data structure. That is, to pass $d_1, \ldots, d_m$, we simply encode this as $\{(\text{“1”}, [d_1]), \ldots, (\text{“}m\text{”}, [d_m])\}$ (where $[\cdot]$ denotes a suitable string representation of the data structure).

The operator implementation itself is a procedural method. From the Turing-completeness of C++ and the fact that we did not make any assumptions about *op*, its arity or the number or types of the additional parameters, it follows that our implementation is complete with respect to the formal operator definition.

(ii) We now show that our framework also allows us to apply operator *op* on (atomic or composed) belief bases $P_1, \ldots, P_n$ with additional parameters $d_1, \ldots, d_m$, i.e., we want to compute

$$op(AS(P_1), \ldots, AS(P_m), d_1, \ldots, d_m)$$

This is done by the following program.

$$\begin{aligned}
P = \{ args(1, H1) &\leftarrow \&hex[P_1, \text{“”}](H1). \\
&\ldots \\
args(n, Hn) &\leftarrow \&hex[P_n, \text{“”}](Hn). \\
kvpairs(\text{“1”}, [d_1]). & \\
&\ldots \\
kvpairs(\text{“}m\text{”}, [d_m]). & \\
result(H) &\leftarrow \&operator[op, args, kvpairs](H).\}
\end{aligned}$$

According to the definition of the HEX semantics in Section 2.3, the external atom &*operator* has assigned a 5-ary function

$$f_{\&operator} : 2^{HB(P)} \times (\Sigma_c)^4 \rightarrow \{true, false\}.$$

where the first argument is a (restricted) interpretation, the second the name of an operator, the third a list of input parameters, the fourth a set of key-value pairs, and the fifth an integer value.

Our implementation of &*operator* as discussed in Section 5.2 makes sure that

$$f_{\&operator}(I, OpName, In, Params, Out) = true$$

if and only if the operator with the name in *OpName*, applied on belief bases *In* and additional arguments *Params* delivers a result that is identified by handle *Out*.

The given program obviously passes the handles $H1, \ldots, Hn$ as well as parameters $d_1, \ldots, d_m$ to the operator *op*. It is clear that the ordering of the passed arguments can be reconstructed from the running index in the first argument of *args* resp. *kvpairs*.

The implementation of the external atom makes sure that the operator's result is uniquely identified by H. This shows that we are able to call arbitrary operators defined within the framework. Since we have also shown that any operator according to Definition 4.7 can be implemented within the framework, it follows that our implementation is complete with respect to that definition.

Correctness. We show that each operator that can be implemented and applied using the framework exactly corresponds to an operator according to Definition 4.7. For the correctness proof, we first restrict the usage of our implementation in the following way. The framework supports passing additional parameters as sets of key-value pairs over strings. For now we assume that this set must be of kind

$$\{(\text{“1”}, str_1), \ldots, (\text{“}m\text{”}, str_m)\}.$$

Let *op* be an operator within the framework. It is provided by a method within a C++ class implementing the interface IOperator. By definition of this method, its parameters are the arity, a vector of HexAnswer (defined as sets of answer sets) and a vector of key-value pairs. Let n be the operator's arity (possibly a variable) and m the number of parameters $d_i \in \mathcal{D}_i$ with $1 \leq i \leq m$

Since there exists a concrete implementation of the operator, it is computable. But then we can also define a function of kind

$$op(SAS_1, \ldots, SAS_n, str_1, \ldots, str_m)$$

where SAS_i are the sets of answer sets passed to the operator and d_i are the additional parameters. Note that we have replaced our set of key-value pairs

$$\{(\text{“1”}, str_1), \ldots, (\text{“}m\text{”}, str_m)\}$$

by the list

$$str_1, \ldots, str_m;$$

but this does not lead to a loss of information since the key was only a running index anyway. The final formulation exactly corresponds to Definition 4.7, which shows that all restricted operators which can be implemented must necessarily respect the theoretical definition.

Finally, we come back to our assumption that the additional parameters are of kind

$$\{(\text{“1”}, str_1), \ldots, (\text{“}m\text{”}, str_m)\}$$

In fact, dropping this restriction does not lead to a gain in expressiveness since each parameter of kind

$$\{(\text{“}k_1\text{”}, str_1), \ldots, (\text{“}k_m\text{”}, str_m)\}$$

can be rewritten to

$$\{(\text{“1”}, \text{“}k_1 = str_1; \ldots; k_m = str_m\text{”})\}$$

(assuming that the characters “ = ” and “;” do not occur as character within the values str_i)

Clearly, the second formulation encodes the same information but does respect our initial restriction.

Since each restricted operator written within the framework corresponds to an operator according to Definition 4.7, and the fact that dropping the restriction does not lead to a gain of expressiveness, it follows that *all* operators that can be implemented in the framework respect the formal foundation of operators necessarily. Therefore our implementation is correct. □

5.3 Common Signature

In Section 4.3 we have introduced two types of incompatibilities. The first one is a consequence of the fact that several belief bases have been developed independently and thus are may syntactically incompatible even if they store semantically equivalent information. The second type arises during the merging task if different sources store contradicting information.

Section 4.4 argues that the first type can be resolved by a *common signature*. That is, sources are made syntactically compatible by using a shared vocabulary.

To become more precise, this means that we need to define a new knowledge representation formalism that is expressive enough to map *any* of the original belief bases onto. In case of HEX programs, this common signature is nothing more than a set of predicates and a set constants (actually, predicates are sufficient as we will see later). This shall be illustrated with an example.

Example 5.3. Consider two belief bases that store employee information. The sources origin from two companies that have been fused. Thus, also their employee databases need to be united. The first source is a relational table and its attributes cover the first and last name, annual salary and phone number. The second one is a logic program consisting of just a set of facts that cover the same attributes and additionally the birth date.

Source 1 - A relational table:

lastname	firstname	salary	phone_number
Amadopolis	Aristotle	350.200,00	+1 555 1119876
Guy	Unnamed	20.700,00	+1 555 1117890

Source 2 - A logic program:

$$
\begin{aligned}
P = \{ & employee(\text{“}C.\ Montgomery\text{”}, \text{“}Burns\text{”}, \text{“}30.400.700,00\text{”}, \\
& \qquad \text{“}0555\ 2229876\text{”}, \text{“}1910-02-24\text{”}). \\
& employee(\text{“}Homer\ J.\text{”}, \text{“}Simpson\text{”}, \text{“}17.990,00\text{”}, \\
& \qquad \text{“}0555\ 3339876\text{”}, \text{“}1960-07-12\text{”}). \\
& employee(\text{“}Lenny\text{”}, \text{“}Leonard\text{”}, \text{“}22.750,00\text{”}, \\
& \qquad \text{“}0555\ 4449876\text{”}, \text{“}1965-06-17\text{”}). \\
& employee(\text{“}Carl\text{”}, \text{“}Carlson\text{”}, \text{“}21.750,00\text{”}, \\
& \qquad \text{“}0555\ 6669876\text{”}, \text{“}1966-01-27\text{”}).\}
\end{aligned}
$$

Obviously the two knowledge sources are syntactically incompatible. They use completely different formalisms to store their content. Additionally, source 2 stores one extra attribute (the birth date) and uses a different format for storing the phone number. While in source 1, each number is prefixed with the country code, source 2 assumes all employees to have domestic providers.

Now we need to define our common signature. The formalism to use is clear. Since we want to work with dlvhex, we need to map both sources to a logic program resp. an answer set. But how can we solve the problem with the extra attribute and the differing number formats? Basically we have two choices. We can either drop the information in source 2 or we can introduce it in source 1 (either by completing the information or by introducing a default value), where the better choice is to keep as much of the data as possible as long as it is expressible in the final formalism. This is called *data completeness*[Naumann and Häussler, 2002]. In case of the phone number, we clearly need to agree upon one of the formats and do a conversion.

Assume that we decide to use a default value for employees with an unknown birth date (n/a) and use the (more general) phone number format with prefixed country codes. Then the following common signature is suitable:

$$
\begin{aligned}
\Sigma_p = \{ & \ employee/5\}, \\
\Sigma_c = \{ & \text{“}Amadopolis\text{”}/0, \text{“}Aristotle\text{”}/0, \text{“}350.200,00\text{”}/0, \text{“}+1\ 555\ 1119876\text{”}/0, \\
& \text{“}Guy\text{”}/0, \text{“}Unnamed\text{”}/0, \text{“}20.700,00\text{”}/0, \text{“}+1\ 555\ 1117890\text{”}/0, \\
& \text{“}C.\ Montgomery\text{”}/0, \text{“}Burns\text{”}/0, \text{“}30.400.700,00\text{”}/0, \text{“}+1\ 555\ 2229876\text{”}/0, \\
& \text{“}1910-02-24\text{”}/0, \\
& \text{“}Homer\ J.\text{”}/0, \text{“}Simpson\text{”}/0, \text{“}17.990,00\text{”}/0, \text{“}+1\ 555\ 3339876\text{”}/0, \\
& \text{“}1960-07-12\text{”}/0, \\
& \text{“}Lenny\text{”}/0, \text{“}Leonard\text{”}/0, \text{“}22.750,00\text{”}/0, \text{“}+1\ 555\ 4449876\text{”}/0, \\
& \text{“}1965-06-17\text{”}/0, \\
& \text{“}Carl\text{”}/0, \text{“}Carlson\text{”}/0, \text{“}21.750,00\text{”}/0, \text{“}+1\ 555\ 6669876\text{”}/0, \\
& \text{“}1966-01-27\text{”}/0\}
\end{aligned}
$$

(numbers following the slashes denote the arities)

Using this signature we can easily express the information of both belief bases. The mapping is shown in Section 5.4. For now it is sufficient to know that a knowledge source over this common signature is exactly what we need.

Note that the common signature needs to be defined separately for each merging task since it strongly depends on the data in the knowledge sources. Thus it needs to be part of the merging plan language.

We come to the conclusion that the common signature definition is simply a set of predicate names paired with their arities (once more: we will show that it is not required to define the constants explicitly). In case of the above example the according section is:

```
[common signature]
predicate: employeeCommon/5;
```

See Appendix A for a formal syntax definition of the common signature section in merging plan files.

5.4 Mappings

In the previous section we introduced common signatures and argued that they need to be expressive enough to represent any of the input belief bases. Now we derive the requirements for a mapping language, which is tightly paired with the definition of the common signature.

Recapitulate the formal definition of mappings given in Section 4.5. A mapping is a function that maps input belief bases over arbitrary signatures, given as sets of answer-sets, onto semantically equivalent set of answer-sets over the common signature. This is illustrated with a continuation of the example concerning employees that was started in Section 5.3. The table in Example 5.4 shows the mapping symbolically.

Example 5.4. A mapping of the two sources in Example 5.3 (symbolically).

$(W, X, Y, Z) \in$ *source 1*	$\Rightarrow$	$employee(X, W, Y, Z, n/a) \in$ *source c*
$(V, W, X, Y, Z) \in$ *source 2*	$\Rightarrow$	$employee(V, W, X, Y, Z') \in$ *source c* where Z' is Z with the leading 0 being replaced by $+1$

The translation of this abstract mapping into HEX program rules is straight forward. Assume

$$\&relational[\text{“}source1\text{”}](W, X, Y, Z)$$

provides access to the relational table of source 1 and $employee(V, W, X, Y, Z)$ for accessing source 2 is defined. Further assume that $phoneConversion(Z, ZZ)$ replaces the leading 0 of phone number Z by the country code $+1$. Then we can define the predicate $employeeCommon(V, W, X, Y, Z)$ as follows.

Example 5.5. A logic program that maps the two sources of Example 5.3 to the common signature.

$$\begin{aligned} P = \{ employeeCommon(X, W, Y, Z, \text{“}n/a\text{”}) &\leftarrow \&relational[\text{“}source1\text{”}](W, X, Y, Z). \\ employeeCommon(V, W, X, Y, ZZ) &\leftarrow employee(V, W, X, Y, Z), \\ & \quad phoneConversion(Z, ZZ).\} \end{aligned}$$

Afterwards the predicate *employeeCommon* provides access to the united information of both knowledge sources.

Now it becomes clear why predicate names are sufficient to define the common signature. The constants are given implicitly by the mapping rules. All constants that occur somewhere in the program, or that are returned from external atoms, are automatically part of the common signature.

We come to the conclusion that mappings are sets of HEX rules. Most of the rules will use elements of the common signature as their head literals. However, there is no restriction about this, since in selected cases it may be reasonable to use axillary predicates.

Note that while we have only *one* common signature definition for a merging task, we need to provide a separate mapping *for each* input source. A complete definition of a mapping includes *(i) the belief base name* and *(ii) a set of mapping rules*. For the above example the final mapping (note that \' is the quoted version of "):

```
[belief base]
name: source1;
mapping: "employeeCommon(X, W, Y, Z, \'n/a\') ←
                        &relational[\'source1\'](W, X, Y, Z).";

[belief base]
name: source2;
mapping: "employeeCommon(V, W, X, Y, ZZ) ←
                         employee(V, W, X, Y, Z),
                         phoneConversion(Z, ZZ).";
```

Listing 5.2: Mapping the programs to the common signature

Additionally to the definition of mappings by embedded rules of type

mapping: "...";

also a second variant was implemented. Using statements of kind

source: "externalprogram.hex";

one can also execute an external program that performs the mapping. This enables the user to separate mapping rules from the rest of the merging task definition. In case an external program is defined, additional mapping rules of the former kind are *not* allowed.

For a formal syntax definition of the mapping section in merging plan files, see again Appendix A.

5.5 Merging Plans

At this point we have defined a common signature and mapped each input belief base to it. We further have a repository of operator definitions. Now we can define the merging plan in the narrower sense. That is, we decide how the sources shall be combined by the operators.

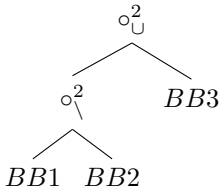
In Example 5.6 have three input sources BB_1, BB_2 and BB_3. First we combine BB_1 with BB_2 using the set difference operator, that we formally define as follows:

$$\circ^2_{\setminus} \quad : 2^{\mathcal{A}(\Sigma^C)} \times 2^{\mathcal{A}(\Sigma^C)} \rightarrow 2^{\mathcal{A}(\Sigma^C)}, \ \textit{where}$$

$$\circ^2_{\setminus} \quad (SAS1, SAS2) = \{AS1 - (\bigcup SAS2) | AS1 \in SAS1\}$$

Informally, the result of this operator is the set of all answer sets of input source 1 restricted to those literals that do not occur in *any* of the answer sets of source 2. The result of this operator is again a set of answer sets that is subsequently united with the result of BB3 (using the semantics defined in Section 4.5).

Example 5.6.



We now need to design an appropriate syntax for defining such merging plans in a machine-readable form. Obviously something hierarchical is needed, thus an XML-like structure is desired. However, XML is clearly over-dimensioned. All we need to store is a tree, where the leaf nodes are names of belief bases and the inner nodes are names of operators. Therefore an easy-to-parse alternative was chosen. The following code snippet shows how the merging plan of the example can be defined using the implemented formalism. For a detailed description of the formal syntax, see once more Appendix A.

```
[merging plan]
{
        operator: union;
        {
                operator: setminus;
                {BB1};
                {BB2};
        };
        {BB3};
}
```

Listing 5.3: Merging plan for Example 5.6

A merging plan can either be composed or atomic. Atomic plans are simply belief bases; composed ones consist of an operator name and n sub plans. In the example, the outermost plan is composed: it applies the operator *union* on two subplans. The first subplan is itself composed (the set difference of *BB1* and *BB2*), whereas the second one is belief base BB3 (atomic).

In case of the employee example, the resulting merging plan is:

```
[merging plan]
{
        operator: union;
        {source1};
        {source2};
}
```

Listing 5.4: Merging plan of the employee example (5.3)

Now we have laid down the syntax of merging plans as part of the merging task input. However, it is clear that dlvhex is not capable of executing this definition directly. Rather it needs to be converted into a pure HEX program. This is the topic of the Section 5.7.

5.6 A Language for Merging Tasks

In the last sections we have shown how to implement operators, the common signature, mappings and merging plans. Now we put all these things together. In other words, we want to define a language for merging task description that is expressive enough to allow the specification of all the elements shown in the overview (see Section 5.1) and at the same time is simple enough for a pleasant usage without further knowledge of the internals.

Belief bases will in general exist long before something like belief merging is even considered. In many cases they are provided by third parties and the representation formalisms have a wide

variety. Hence, it is not necessary or even desirable to make any assumptions about them in the merging plan language. The only thing we know is that there are abstract belief sources written in some formalism.

Operators are clearly used in merging plans. Nevertheless their semantics is independent of certain merging tasks. It would be a waste of resources if one redefined well-known operators in each plan. Additionally this would bear the risk of inconsistencies in the sense that intentionally equivalent operators behave differently in different merging tasks due to implementation errors or different understandings of the programmers. Thus it is much more desirable to have a *pool of operators* that is just used in a merging task definition but not part of it.

Thus, what remains to be included in the merging plan language is a triple consisting of the common signature, a mapping of belief bases onto the common signature and the merging plan in the narrower sense. We have already shown the implementation of each of them in the previous sections. The total description of a merging task is just a file consisting of all these elements. For the employee example, the result is shown in Listing 5.5.

```
[common signature]
predicate: employeeCommon/5;

[belief base]
name: source1;
mapping: "employeeCommon(X, W, Y, Z, \'n/a\') ←
                        &relational[\'source1\'](W, X, Y, Z).";

[belief base]
name: source2;
mapping: "employeeCommon(V, W, X, Y, ZZ) ←
                        employee(V, W, X, Y, Z),
                        phoneConversion(Z, ZZ).";

[merging plan]
{
        operator: union;
        {source1};
        {source2};
}
```

Listing 5.5: Merging file of the employee example (5.3)

What remains to do is to provide a translation mechanism of this description into a pure HEX program such that it can actually be executed.

5.7 Translating Merging Plans

The translation of merging plans into HEX programs is schematically explained in Figure 5.4 and 5.5.

Since the translation is done inductively, we will first look at atomic subplans. In Figure 5.4 a belief base called *bb* with its mappings is translated into the code nearby. We simply generate a call of the external atom &*hex* (introduced in Section 3) with the rules from the mapping. The result is a handle to the answer of this belief base *mapped to the common signature*, which is stored in $sources(bb, X)$. Note that in fact we have created code for the *mappings* rather than for the *belief bases*. But remember that we said belief bases will already exist before merging is done. Thus it is not necessary to create code for them. It is sufficient to write mappings that query them.

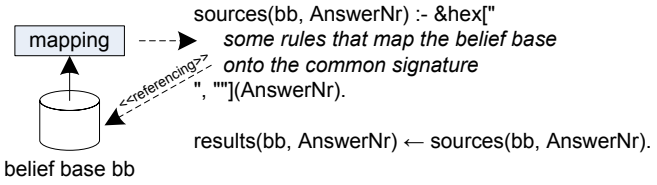


Figure 5.4: Translation of atomic merging plans

The last line of code is just a renaming of *sources* to *results*. While the predicate *sources* is only used for handles to the answers of belief bases, *results* is more general since it is also used for handles to interim results (see below).

Proposition 2. The translation shown in Figure 5.4 derives a handle to a cache entry that exactly corresponds to the evaluation of an atomic merging plan (a belief base) according to Definition 4.10.

Proof. Our implementation of &*hex*, as discussed in Chapter 3, executes the nested HEX program passed as string literal and returns a unique handle *AnswerNr* to its result.

Since the nested program is given in form of the mapping rules provided by the user, such that they query the external sources in the bodies and derive atoms over the common signature in the heads, its result corresponds to $\mu_i(AS(P_i))$, where P_i is a belief base and μ_i its mapping. But this is equivalent to our definition of the semantics of an atomic merging plan, concluding the proof, apart from the fact that our translation delivers a *handle* to the answer sets, rather than the answer sets themselves, see below. □

Now we will look at composed plans. Assume that a fictitious operator □ is applied to subplans 1 to n. The subplans can be either atomic or composed in turn. We assume that their results have already been computed and are accessible through $results(sp_1, \cdot)$ to $results(sp_n, \cdot)$. In other words, our induction hypothesis is that we can access the result of an arbitrary sub-merging plan.

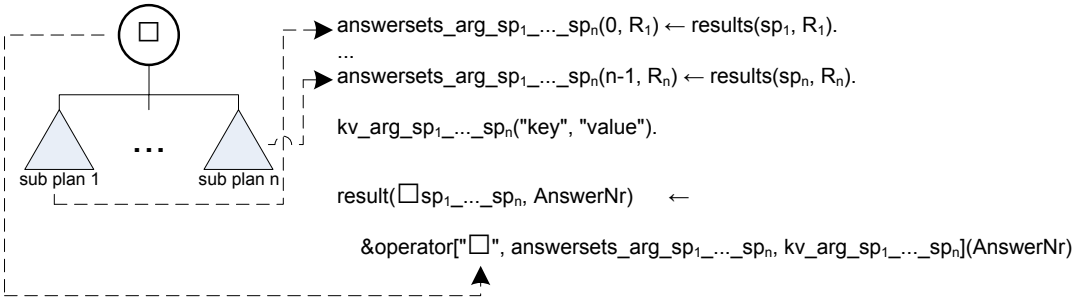


Figure 5.5: Translation of composed merging plans

Now we show how we can generate code that delivers the result of the outermost operator application. First recapitulate the definition of the external atom &*operator* from Section 5.2. It expects as its second parameter a predicate that lists all the handles which shall be passed to the operator. Thus we need to construct such a predicate. $answersets\_arg\_sp_1\_\ldots\_sp_n$ is a binary predicate with a running index i as its first argument and a handle H as its second, that shall be passed to the operator as the i-th parameter.

The confusing name of this predicate is basically a concatenation of the names of the subplans being passed to the operator. This ensures that the predicate name is unique within the generated program.

Next, the predicate $kv\_arg\_sp_1\_\ldots\_sp_n(\text{“}key\text{”}, \text{“}value\text{”})$ is simply a set of key-value pairs where additional (non-answer set) parameters can be passed if this is necessary due to the operator's semantics (see IOperator in Section 5.2).

Finally &*operator* can be called. Its first parameter is the name of the operator that shall be called, its second is the predicate that lists all the handles to pass (the one with the strange name) and its third one the list of key-value pairs. The result is a handle to the operator's answer. This handle can now be used in the same way as we used the handles to the belief bases. It can be passed to a further operator call or it can act as final result (see below).

Proposition 3. The translation shown in Figure 5.5 derives a handle to a cache entry which exactly corresponds to the evaluation of composed merging plans according to Definition 4.10.

Proof. The proof is by induction on the maximum number of nested operators. Our induction base is the correctness of Proposition 2, i.e., the translation of atomic belief bases.

Induction step: For each atom $results(sp_i, R_i)$, R_i is a handle to the result of sub-merging plan i by induction hypothesis (since the nesting level of this plan is smaller). Clearly the tuples over predicate $answersets\_arg\_sp_1\_\ldots\_sp_n$ correctly encode the ordering of the arguments passed to the operator. Furthermore, atoms over $kv\_arg\_sp_1\_\ldots\_sp_n$ correctly encode the key-value pairs passed to the operator.

Therefore, by Proposition 1, which states the correctness of our operator implementation with respect to the semantics laid down in Definition 4.7, the last rule in the translation delivers the result of the currently applied operator. And this corresponds to the semantics of a composed merging plan according to Definition 4.10 (once more, apart from the fact that our translation delivers a *handle* to the answer sets, rather than the answer sets themselves, see below).

This shows that the translation of arbitrarily nested merging plans exactly corresponds to the semantics of merging plans. □

Corollary 1. Since the semantics of a formal merging task is nothing more than the result of the topmost merging operator (see Definition 4.10), it follows immediately from Proposition 3 that the output of the program generated by the above translation rules delivers a handle to the final result.

Thus, we have shown that we can translate atomic merging plans into semantically equivalent HEX programs. We further have inductively shown that we can translate an operator application whenever we have handles to the results of the operator's subplans. In total this means that we can translate *any* merging plan into pure HEX programs.

However, at this point we have a *handle* to the answer of the topmost operator application. It would be nice if this handle could be mapped to the *“real answer”* of the resulting HEX program. This is the gist of the next section.

5.8 Extracting the Final Answer

In the last section we have translated merging plans of arbitrary depth into pure HEX programs. Clearly, the result of the outermost operator application is the final result of the whole merging plan.

Copying Information from Internal Structures

Until now we have just worked with *handles* to answers. That is, the result contains a literal like $result(op, 4)$. The actual content of this result is hidden within the internal data structures of the nested HEX module of the mergingplugin.

Of course we could use the external atoms &*answersets*, &*predicates* and &*arguments* to look into the result, but the more natural and thus more desirable solution would clearly be to map the internal results somehow to the *real* result of the program.

And this is exactly the reason why we needed to define the common signature in Section 5.3. If we take a closer look at the translation of merging plans as we defined it until now, we realize that we *never* referenced the common signature so far. Sure, the mapping rules used it in the rule heads. But this does not require the predicates to be listed in a separate section of the merging task input. We could also simply use certain predicates without declaring them explicitly.

But if we want to *extract* the internal answer sets that a handles refers to, we need to know which predicates are of interest. Let's assume that the handle to the result of the topmost operator application is named $finalResult(AnswerNr)$. Since we can map only one answer set at a time, assume further that the predicate $selectedas(AnswerSetNr)$ is *true* for exactly one (valid) answer set handle (for details see below). Then Figure 5.6 shows how it can be mapped to the actual answer of the HEX program.

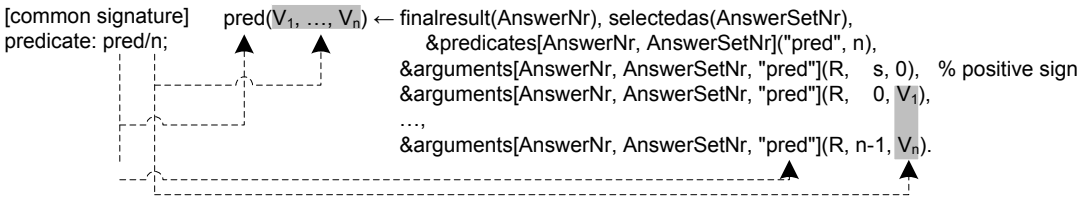


Figure 5.6: Extraction of Answer Sets

We need two such rules per predicate, namely for the positive and the strongly negated version. In the body of the rule we fix the handle to the answer of the outermost operator, called $finalresult(AnswerNr)$ as well as the handle to the currently selected answer set, which is denoted as $selectedas(AnswerSetNr)$. We further check if the predicate name actually occurs within this answer set by evaluating the &*predicates* atom with the according parameters.

Then we can extract the n parameters for an n-ary predicate (plus its sign). For that purpose the &*arguments* atom is evaluated n times, each time with the same input parameters (namely the handles to the answer and the answer set as well as the predicate name). The output is filtered as follows. We pass a variable as first output parameter. Remember that this is a running index that has no inherent meaning but that it can be used to figure out which of the arguments belong to one occurrence of the predicate (see Section 3.5). Therefore we pass the same variable in each call to make sure that we retrieve the arguments that belong together.

The second and third parameter are used to determine the values on parameter positions 0 to $n-1$. We store these values in the variables V_1 to V_n. Finally we can derive the according literal in the rule head. This is illustrated with a continuation of the employee example.

Additionally we include a check of the sign (either 0 for positive or 1 for strongly negated). In a second rule of the same kind, we check for negative sign and add a negation in the rule head.

Example 5.7. This code is generated by the merging plan compiler for the employee example. It extracts the contents of the final result and maps it to the program's answer.

$$
\begin{aligned}
P = \{ & employeeCommon(V_1, V_2, V_3, V_4, V_5) \leftarrow \\
& \&finalresult(AnswerNr), selectedas(AnswerSetNr), \\
& \&predicates[AnswerNr, AnswerSetNr](\text{“}employee\text{”}, 5), \\
& \%\ positive \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, s, 0), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 0, V_1), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 1, V_2), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 2, V_3), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 3, V_4), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 4, V_5). \\
\neg & employeeCommon(V_1, V_2, V_3, V_4, V_5) \leftarrow \\
& \&finalresult(AnswerNr), selectedas(AnswerSetNr), \\
& \&predicates[AnswerNr, AnswerSetNr](\text{“}employee\text{”}, 5), \\
& \%\ negative \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, s, 1), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 0, V_1), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 1, V_2), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 2, V_3), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 3, V_4), \\
& \&arguments[AnswerNr, AnswerSetNr, \text{“}employee\text{”}](R, 4, V_5).\}
\end{aligned}
$$

Selecting Answer Sets

We used the predicate $selectedas(AnswerSetNr)$ without explaining in detail how it is defined. To illustrate the sense of this predicate, just imagine what happens if we simply drop it from the above definition.

In case of the employee example, the final result would not change. But this is only the case because the program delivers a unique answer set. However, if the outermost operator delivers a result consisting of multiple answer sets, they simply would be put them into one during extraction.

Thus we need to ensure that our program produces n answer sets, namely those that are generated by the outermost operator. This means, we copy only the information from the *currently selected* one. This is easily realizable as shown in the following program:

$$
\begin{aligned}
P_{select\_as} = \{ & \mathit{finalanswersets}(\mathit{AnswerNr}, \mathit{AnswerSetNr}) && \leftarrow && \mathit{finalresult}(\mathit{AnswerNr}), \\
& && && \&\mathit{answersets}[\mathit{AnswerNr}] \\
& && && \quad (\mathit{AnswerSetNr}). \\
& \mathit{selectedas}(\mathit{AnswerSetNr}) \vee \neg \mathit{selectedas}(\mathit{AnswerSetNr}) && \leftarrow && \mathit{finalanswersets} \\
& && && \quad (\mathit{AnswerNr}, \mathit{AnswerSetNr}). \\
& && && \% \textit{ Select at least one} \\
& \mathit{atleastoneasselected} && \leftarrow && \mathit{selectedas}(\mathit{AnswerSetNr}). \\
& && \leftarrow && \text{not } \mathit{atleastoneasselected}. \\
& && && \% \textit{ Select at most one} \\
& && \leftarrow && \mathit{selectedas}(\mathit{AnswerSetNr1}), \\
& && && \mathit{selectedas}(\mathit{AnswerSetNr2}), \\
& && && \mathit{AnswerSetNr1} \neq \mathit{AnswerSetNr2}. \}
\end{aligned}
$$

The first line extracts the number of answer sets that are available in total. Then, each of them can either be selected or not selected. The remaining part of the code ensures that *exactly one* of them is selected at one time. First, we introduce a propositional variable that checks if at least one is selected, and prevent that this is not the case (using a constraint). Then we further prevent that more than one is selected.

Thus, in total we have selected exactly one answer set X, and *selectedas* is *true* only for the handle to X and *false* for all other handles $Y \neq X$. Together with the code in Section 5.8 we extract one answer set after the other and copy its content into the answer set of the host HEX program.

This concludes the extraction of the final result, such that the answer of the translated program will contain exactly the answer sets which were computed by the topmost operator in the merging plan. This is demonstrated by a final example.

Example 5.8. Consider the following merging input, where the union operator is defined as in Section 4.5, i.e., it computes the pairwise combinations of answer sets.

```
[common signature]
predicate: a/0; predicate: b/0; predicate: c/0; predicate: d/0;

[belief base]
name: kb1;
mapping: "a.";

[belief base]
name: kb2;
mapping: "b v d. c ← b.";

[merging plan]
{
        operator: union;
        {kb1};
        {kb2};
}
```

Then the result of the program that is obtained by applying the described translation rules, has two answer sets: $\{a, b, c\}$ and $\{a, d\}$.

Proposition 4. The overall translation shown in Figures 5.4, 5.5 and 5.6 and program P_{select_as} derives a set of answer sets that correspond to the semantics of a merging task according to Definition 4.10.

Proof. By correctness of Proposition 3 and Corollary 1, we know that the translation shown in Figures 5.4 and 5.5 delivers a HEX program with an answer set containing a handle to an internally cached result of the topmost operator in the merging plan. This result is exactly the semantics of a merging task by definition.

What remains to by shown is that the translation shown in Figure 5.6 correctly dereferences the handle by mapping this internal data structure to the actual answer sets of the generated program.

We first discuss the special case that the topmost operator returns an empty set of answer sets. In this case, the atom *atleastoneselected* cannot be derived, which violates the according constraint. Therefore the overall program is inconsistent, i.e., it has no answer sets. But this is exactly what we expect since it is equivalent to the return value of the topmost operator by assumption.

Now we can safely assume that the topmost operator returns at least one answer set. We will prove that the translation is also correct in this case. This is shown independently for each relevant predicate, i.e., each predicate of the common signature. Suppose we are currently extracting the atoms over predicate "pred" of arity n. Since $selectedas(AnswerNr)$ is *true* for exactly one answer set (using a typical guess rule), and there does exist at least one answer set by assumption, we correctly extract the content of one answer set at one time, which avoids mixing up the contents. Then it is obvious to see that the translation extracts the n arguments of each occurrence of "pred" as well as its sign. The rule head correctly derives an atom over the current predicate with the same arguments as in the internal cache, but now as member of the actual answer set.

If we introduce two rules of this kind for each predicate, one for positive and one for strongly-negated atoms, this obviously extracts all relevant *literals* from the result of the topmost operator and maps it onto the answer sets of the program. Further observe that by definition of our translation rule, only atoms over the common signature are extracted. But then the answer sets of the program exactly correspond to the formal semantics of merging plans according to Definition 4.10. □

5.9 Merging Plan Compiler

The translation strategy that was explained in the previous section is finally implemented such that it can be done automatically. And this is strongly connected with the actual benefit of the framework. The user only has the specify the merging plan; the translation into a semantically equivalent program that organizes the information flow is done completely automatically.

For this purpose a command-line tool called mpcompiler was implemented that is installed as part of the mergingplugin. A typical usage will be:

```
mpcompiler mergingplan1.mp | dlvhex --
```

First the merging plan is translated into a HEX program, that is then passed to dlvhex. For a complete user guide of the mpcompiler, see Appendix B.

Good judgement comes from
experience. Experience comes
from bad judgement.

Jim Horning

Chapter 6

Application Scenarios

6.1 Implementing a Dalal-style Operator

Before we look at concrete practical examples, we define a more advanced operator than in the previous sections to demonstrate the features of the framework. According to [Gabbay et al., 2009] this is the most often used merging operator in the literature. It was already briefly discussed in Section 4.2.

Note that *Dalal's operator* is neither an official term nor does it uniquely define a merging strategy. What [Dalal, 1988] actually introduced is not an operator but a distance function for comparing two interpretations. To become more precise, it computes the Hamming distance between the interpretations, i.e., it counts the conflicting atoms.

Subsequently, these pairwise distances can be aggregated and used to measure the dissimilarity between interpretations and knowledge bases, and finally between interpretations and sets of knowledge bases. The details leave room for parameterization. This overall distance value can then be used to define a merging operator by simply choosing the merged belief base such that its models have minimum overall distance to the sources, as this is done for instance by [Gabbay et al., 2009].

Moreover, using the Hamming distance for comparing interpretations, as this was done by Dalal, is by far not the only possibility to measure distances. As we will see, the approach can be generalized by parameterizing it on three levels in total. Nevertheless we will use the term *Dalal-style operator* to refer to the collection of operators that can be defined on top of a generalized version of his distance function.

Model-theoretic Formulation

We first look at the model-theoretic formulation since most papers assume the belief bases to be just sets of propositional or first-order formulas rather than logic programs. So let

$$K = (KB_1, \ldots, KB_n)$$

be our vector of belief bases and let further $\mathcal{M}$ be the set of all propositional interpretations.

Defining an operator for merging these n sources into one is done by implementing the following three level approach.

1. Defining a distance function for comparing two interpretations
2. Defining a distance function for comparing an interpretation to a belief base
3. Defining a distance function for comparing an interpretation to our vector of belief bases

Each step builds on top of the previous one. After this has been done, it is straightforward to use this distance measurement for computing a merged belief base with minimal distance to the sources, i.e., to define a merging operator upon the distance between interpretations and vectors of belief bases.

Comparing two Interpretations

First we need to introduce a function for distance measurement between two interpretation. We call this function $\underline{d}$ for *distance*.

Definition 6.1. A distance function $\underline{d}(I, J)$ for comparing an interpretation I and an interpretation J is of form:

$$\underline{d}(I, J) : \mathcal{M} \times \mathcal{M} \rightarrow \mathbb{R}_0^+$$

such that the following side constraints are satisfied. First, the non-negative distance function is symmetric:

$$\underline{d}(I, J) = \underline{d}(J, I) \ \forall I, J \in \mathcal{M}$$

and second, only equal interpretations have distance 0:

$$\underline{d}(I, J) = 0 \ \textit{iff} \ I = J$$

Comparing an Interpretation to a Belief Base

Function $\underline{d}$ allows us to define another distance function that computes the dissimilarity between an interpretation and a belief base. For this purpose we simply compute the distance between our reference interpretation and each model of the belief base, and aggregate these values in one way or the other.

Definition 6.2. The distance $d_{\underline{d}}(I, KB_i)$ between an interpretation I and a belief base KB_i can be computed by a function of form:

$$d_{\underline{d}} : \mathcal{M} \times 2^{\mathcal{M}} \rightarrow \mathbb{R}_0^+$$

where the first argument is our reference iterpretation I, and the second one is the set of models of KB_i.

Internally, $d_{\underline{d}}$ will usually evaluate the function $\underline{d}$ several times to compute the final result.

Example 6.1. A very popular choice for $d_{\underline{d}}(I, KB_i)$ is:

$$d_{\underline{d}}(I, KB_i) = \min_{J \in Mod(KB_i)} \underline{d}(I, J)$$

Informally speaking, we take the *best suiting* model of the knowledge base and compute its distance to I. For this purpose we use our formerly defined function $\underline{d}$.

This gives us n distance values between I and each of the belief bases in K. We finally aggregate these values into a single one by applying another function.

Comparing an Interpretation to a Vector of Belief Bases

Definition 6.3. Function $D_{\underline{d},d}(I,K)$ computes the distance between an interpretation I and a vector of knowledge bases K and is defined as

$$D_{\underline{d},d}(I,K) = D(d_{\underline{d}}(I,KB_1),\ldots,d_{\underline{d}}(I,KB_n))$$

where D is an aggregation function of form $D:\left(\mathbb{R}_0^+\right)^n \to \mathbb{R}_0^+$.

Popular choices for $D_{\underline{d},d}(I,K)$ are the sum, the average and the maximum of the individual distances.

Defining a Merging Operator

Finally we can define merging operator $\Delta^n_{\underline{d},d,D}(K)$ on top of our distance function $D_{\underline{d},d}(I,K)$. We just take all consistent interpretations that have minimal overall distance to K.

Definition 6.4. The merged belief base over sources $K = (KB_1,\ldots,KB_n)$ is computed by Dalal's distance operator $\Delta^n_{\underline{d},d,D}$ as follows.

$$\Delta^n_{\underline{d},d,D}(K) = \mathit{form}(\arg\min_{I:I\not\models\bot} D_{\underline{d},d}(I,K))$$

where *form* constructs a belief base from an interpretation I such that $Mod(\mathit{form}(I)) = \{I\}$.

This definition is still incomplete since we need to define what to choose for $\underline{d}$, d and D. These details are application dependent.

The most common choice is $\Delta_{h,m,\Sigma}$ [Gabbay et al., 2009], where

$$h(I,J) = |\{p|I(p) \neq J(p)\}|$$

is the Hamming distance [Dalal, 1988] between two interpretations and

$$m_h(I,KB_i) = \min_{J\in Mod(KB_i)} h(I,J)$$

is our function from above. $h(I,J)$ just computes the number of propositional atoms that are *true* under one and *false* under the other interpretation. Our aggregation function is Σ and computes the sum of the distances to the different belief bases:

$$D_{h,m}(I,K) = \Sigma_{i=1}^n m(I,KB_i)$$

This definition implies that each belief base has the same priority. This operator is now demonstrated with an example.

Example 6.2. We assume our belief bases to be:

$$\begin{aligned} KB_1 &= \{a.,\ \neg b.,\ c \leftarrow a.\} \\ KB_2 &= \{\neg a.,\ d.\} \end{aligned}$$

Then the (relevant) models of the sources are:

$$\begin{aligned} Mod(KB_1) &= \{\{a,c\},\{a,c,d\}\} \\ Mod(KB_2) &= \{\{d\},\{b,d\},\{c,d\},\{b,c,d\}\} \end{aligned}$$

We now need to select the models of the merged belief base. This must be one that has minimal overall distance. In total there are $2^4 = 16$ interpretations that are relevant because we have four atoms, where each can either be *true* or *false*.

We compute the distances as follows.

interpretation	distance to KB_1	distance to KB_2	overall distance to K
$I_1 = \{\}$	$\underline{d}(I_1, KB_1) = 2$	$\underline{d}(I_1, KB_2) = 1$	$D^d(I_1, K) = 3$
$I_2 = \{a\}$	$\underline{d}(I_2, KB_1) = 1$	$\underline{d}(I_2, KB_2) = 2$	$D^d(I_2, K) = 3$
$I_3 = \{b\}$	$\underline{d}(I_3, KB_1) = 3$	$\underline{d}(I_3, KB_2) = 1$	$D^d(I_3, K) = 4$
$I_4 = \{c\}$	$\underline{d}(I_4, KB_1) = 1$	$\underline{d}(I_4, KB_2) = 1$	$D^d(I_4, K) = 2$
$I_5 = \{d\}$	$\underline{d}(I_5, KB_1) = 3$	$\underline{d}(I_5, KB_2) = 0$	$D^d(I_5, K) = 3$
$I_6 = \{a, b\}$	$\underline{d}(I_6, KB_1) = 2$	$\underline{d}(I_6, KB_2) = 2$	$D^d(I_6, K) = 4$
$I_7 = \{a, c\}$	$\underline{d}(I_7, KB_1) = 0$	$\underline{d}(I_7, KB_2) = 2$	$D^d(I_7, K) = 2$
$I_8 = \{a, d\}$	$\underline{d}(I_8, KB_1) = 2$	$\underline{d}(I_8, KB_2) = 1$	$D^d(I_8, K) = 3$
$I_9 = \{b, c\}$	$\underline{d}(I_9, KB_1) = 2$	$\underline{d}(I_9, KB_2) = 1$	$D^d(I_9, K) = 3$
$I_{10} = \{b, d\}$	$\underline{d}(I_{10}, KB_1) = 4$	$\underline{d}(I_{10}, KB_2) = 0$	$D^d(I_{10}, K) = 4$
$\mathbf{I_{11}} = \{\mathbf{c}, \mathbf{d}\}$	$\underline{d}(I_1 1, KB_1) = 1$	$\underline{d}(I_{11}, KB_2) = 0$	$\mathbf{D^d(I_{11}, K) = 1} \Leftarrow$
$I_{12} = \{a, b, c\}$	$\underline{d}(I_{12}, KB_1) = 1$	$\underline{d}(I_{12}, KB_2) = 2$	$D^d(I_{12}, K) = 3$
$I_{13} = \{a, b, d\}$	$\underline{d}(I_{13}, KB_1) = 3$	$\underline{d}(I_{13}, KB_2) = 1$	$D^d(I_{13}, K) = 4$
$\mathbf{I_{14}} = \{\mathbf{a}, \mathbf{c}, \mathbf{d}\}$	$\underline{d}(I_{14}, KB_1) = 0$	$\underline{d}(I_{14}, KB_2) = 1$	$\mathbf{D^d(I_{14}, K) = 1} \Leftarrow$
$I_{15} = \{b, c, d\}$	$\underline{d}(I_{15}, KB_1) = 3$	$\underline{d}(I_{15}, KB_2) = 0$	$D^d(I_{15}, K) = 3$
$I_{16} = \{a, b, c, d\}$	$\underline{d}(I_{16}, KB_1) = 2$	$\underline{d}(I_{16}, KB_2) = 1$	$D^d(I_{16}, K) = 3$

In total there exist two interpretations with minimal distance 1. Therefore the merged belief base $M = \Delta^n_{h,m,\Sigma}(K)$ can by anything such that $Mod(M) \subseteq \{\{c, d\}, \{a, c, d\}\}$.

Recall that operator $\Delta^n_{\underline{d},d,D}(K)$ can be parameterized on three levels: $\underline{d}$ for comparing two interpretations, d for comparing interpretations to belief bases and D for aggregating the individual distances. This illustrates the usefulness of the framework. As we will see, a modification of any of these parameters requires only minor modifications in the merging task definition, but no manual reimplementation from scratch.

Answer Sets as Belief Sets

We now switch the point of view and consider answer sets as belief sets, where belief sets are either the semantics of programs or the result of prior operator applications. This allows us to implement the operator from the previous section in the belief merging framework.

All we need to do is to select a set of literals that has minimum distance to our input answer sets. Since the sources are given as logic programs anyway, a very elegant solution is the use of *weak constraints* to solve this optimization problem. Weak constraints are constraints that may be violated by an answer set, but one has to pay certain costs if this is the case. This allows us to select the *best* interpretation, i.e., the one that causes the least number of constraint violations.

The implementation works as follows. First we collect all the atoms that occur somewhere in the source programs. Let's denote this as

$$Atoms(\pi) = \bigcup_i Atoms(P_i)$$

where π is our vector of programs, as usual. We rename all atoms such that no pair of input programs has atoms in common, i.e., we standardize the programs P'_i s.t.:

$$Atoms(P'_i) \cap Atoms(P'_j) = \emptyset \ \forall i \neq j$$

If $a \in Atoms(P_i)$ is an atom over the original signature, we denote $a[i]$ as the corresponding renamed atom in program P_i'.

Finally we union all P_i' and add the following rules to the program: $P = \bigcup_i P_i' \cup R$ with

$$\begin{aligned} R = \{ & a \vee \neg a. \\ & \leftsquigarrow \quad a[i], \text{not } a. \\ & \leftsquigarrow \quad \neg a[i], \text{not } \neg a. \mid \forall a \in Atoms(\pi)\} \end{aligned}$$

($\leftsquigarrow$ denotes weak constraints).

The first rule makes sure that each atom is either set to *true* or to *false*. This is necessary because of the minimality condition in answer-set programming. Without this rule, an atom a would never be set to any of the truth values because it does not occur as fact anymore after the renaming has been done. Clearly, this makes us lose ternary answer semantics. However, since we only speak about atoms that occur in the answer sets of at least one source, we have evidences for its truth value anyway, so the result *unknown* would never occur.

The two weak constraints try to set the final atom a to the same truth value as the corresponding one $a[i]$ in program P_i'. Since the input programs can be inconsistent when united, this may not always be possible. The reasoner will try to minimize the number of such differences.

Actually the implementation is a bit more complicated than depicted here. We skipped details like dealing with more than one answer set. This would require some more rules to be added. But the ideas should be clear at this point.

Enhancements

The proposed operator can be enhanced in several ways.

Elimination of Auxiliary Predicates

A very natural and reasonable extension is to ignore auxiliary predicates in the source programs when the merged belief base is computed. This can be done by a kind of projection. Formally this means that

$$Atoms(\pi) = \bigcup_i Atoms(P_i)$$

is replaced by

$$Atoms^A(\pi) = \bigcup_i (Atoms(P_i) \setminus Aux(P_i))$$

where $Aux(P_i)$ is specified by the user and denotes the predicates that are used during computation but are irrelevant in the final result. Consequently program R (as defined above) is redefined such that it does *not* contain rules for these auxiliary predicates:

$$\begin{aligned} R^A = \{ & a \vee \neg a. \\ & \leftsquigarrow \quad a[i], \text{not } a. \\ & \leftsquigarrow \quad \neg a[i], \text{not } \neg a. \mid \forall a \in Atoms^A(\pi)\}. \end{aligned}$$

Then

$$P = \bigcup_i P_i' \cup R^A$$

is the new program that needs to be solved by the operator.

User-defined Integrity Constraints

A further extension is the incorporation of user-defined *integrity constraints* that must be satisfied in the resulting base. In the model-theoretic understanding this can be formalized as follows. We will also call them *side constraints*.

Definition 6.5. The merged belief base over sources $K = (KB_1, \ldots, KB_n)$ and integrity constraints C is computed by operator $\Delta^n_{\underline{d},d,D}$ as follows.

$$\Delta^n_{\underline{d},d,D}(K, C) = form(\arg \min_{I: I \not\models \bot, I \models C} D_{\underline{d},d}(I, K))$$

Compare this with Definition 6.4, where the only difference is that now we select the model among all interpretations that are not only consistent, but which also satisfy C.

In terms of logic programming, this can be implemented by redefining the program

$$P = \bigcup_i P'_i \cup R$$

as

$$P^S = \bigcup_i P'_i \cup R \cup S$$

where S is a program consisting of all integrity constraints that need to be satisfied.

Other Aggregation Variants

Recall our basic definition of Dalal's distance operator from Section 6.1:

$$\Delta^n_{\underline{d},d,D}(K) = form(\arg \min_{I: I \not\models \bot} D_{\underline{d},d}(I, K))$$

We said that a concrete operator is influenced by the selection of appropriate functions for the distance measurement $\underline{d}$ and the aggregation D. In the previous sections we used the Hamming distance and the sum which are the most frequently used choices.

Nevertheless also other variants have been discussed in the literature. For instance the minimax rule in [Konieczny and Pino-Pérez, 1998]. Instead of minimizing the sum of distances to the single belief bases, the maximum distance (over all belief bases) is minimized, formally:

$$M(I, K) = \max_i d_{\underline{d}}(I, KB_i)$$

Whereas the sum leads to a kind of majority voting mechanism, this operator makes sure that none of the belief bases diverges too much from the merged one.

A refinement of this aggregate function primarly minimizes the maximum distance (as *minimax*), but in case of equal maximum distance, it minimizes the second highest distance and so on.

Generally, selecting an overall distance function $\Delta^n_{\underline{d},d,D}$ is a matter of comparing sets of interpretations to each other. This problem is related to the *Hausdorff distance* in topology, cf. for instance [Henrikson, 1999].

This concludes our introduction and implementation of Dalal's approach. Now we look at a concrete application.

6.2 Judgment Aggregation

Judgment aggregation is a topic from the research field of *social choice theory*. Nevertheless it is well-suited as a practical application scenario for the merging operator introduced in the previous section, since some authors have observed elementary interrelationships between the two fields. See for instance [Pigozzi, 2006]. It has applications in the areas of economics, political science, philosophy and many others [List and Puppe, 2007]. Basically it deals with the combination of judgments from multiple individuals in order to get a decision of the whole group. Intuitively understandable examples include democratic polls and the aggregation of beliefs of several court-members into a final decision.

Some scenarios require more than a simple majority voting rule. This is demonstrated with a court case, where jurors usually have to answer several yes/no questions (that are formulated by professional judges) *independently*, that will then influence the final decision [Dietrich and Mongin, 2010]. For the sake of simplicity, we assume that we have a court case where only two questions need to be answered:

(V) Is the contract valid?

(B) Was the contract broken?

The party sued will have to pay compensation due (D) if and only if *both* questions are answered with yes. In other words, the individual decisions satisfy the constraint:

$$(V \wedge B) \leftrightarrow D$$

This rule is called *side constraint* or *integrity constraint*. If an individual satisfies it, we also say that it acts *rationally*.

We assume that first each juror has to answer if he or she accepts (V) and/or (B). Then the acceptance of (D) is predefined due to this rule. Finally the answers for (V), (B) and (C) are aggregated by majority decision. Even though this is a very simple voting system, the combination of the meanings is tricky since a problem named *discursive dilemma* (Condorcet's Paradox) can occur [Pettit et al., 2001]. Informally this states that even though each individual respects the rationality rule, this is not necessarily the case for the group decision.

Assume that we have three jurors and consider the following results:

	(V)	(B)	(D)
juror 1	yes	yes	yes
juror 2	no	yes	no
juror 3	yes	no	no
group decision	yes	yes	no

Each juror respects the integrity constraint, that D is *true* iff V and B are accepted, but the final decision denies (D) even though it accepts both (V) and (B).

Usual Solutions

Traditionally we can overcome this problem by two strategies called *premise-based procedure* and *conclusion-based procedure* [Pigozzi, 2006].

In the premise-based approach, we apply the majority rule *only* for the premises (V) and (B). The conclusion (D) is then entailed using the rationality rule and the majority votes for the premises, which leads to the final answer "yes" in this example.

In contrast, the conclusion-based solution applies the majority rule *only* for the conclusion (D), which gives us the final answer "no". This is summarized in the following table.

	(V)	(B)	(D)
juror 1	yes	yes	yes
juror 2	no	yes	no
juror 3	yes	no	no
premise-based	yes	yes	$\Rightarrow$ yes
conclusion-based			no

Both strategies have drawbacks. Despite from the fact that they lead to different group decisions, conclusion-based aggregation does only state if (D) is accepted or denied, but does not justify this. A court case is a brilliant example where this is a major lack, since the parties have the right to know the reasons for the final decision.

In contrast to that, premise-based methods make people worry if they can actually control the collective decision [Pigozzi, 2006].

Task Formalization

Before we continue with resolution strategies, we define the task formally.

Let $\mathcal{S}$ be a set of propositional statements. Then an individual judgment $J_i \subseteq \mathcal{S}$ is the set of all statements accepted by agent i. The n-tuple $J = (J_1, \ldots, J_n)$ is called *profile*.

Further let C be (a conjunction of) our integrity constraints and $J_i \models C \; \forall 1 \leq i \leq n$. Then an aggregate function f maps the profile to the collective decision:

$$f : \mathcal{S}^n \mapsto \mathcal{S}$$

$$J_A = f(J)$$

s.t.

$$J_A \models C$$

for each profile J. Note that in general it is *not* a requirement that the individual decisions satisfy the integrity constraints. It could also be an interesting task to compute a rational aggregate judgment, even though individuals may act irrationally. Nevertheless in most settings one can expect rational individuals, which is denoted as *universal domain*, see next subsection.

An Impossibility Result

One could argue that these troubles are a lack of the proposed majority rule that it is applied independently for (V), (B) and (D). But it turns out that the same problem can arise also with other aggregate methods.

Even worse, it can be shown that if we insist on several reasonable properties of an aggregation function, there does not exist one that does not lead to irrational group decisions. This is now described more formally.

We define the following properties for aggregation functions f:

U *Univeral domain*
Any logically possible individual judgment (i.e. $J_i \models C$) is accepted as input by the aggregate procedure.

R *Collective Rationality*
If J is a profile and C are integrity constraints, then $f(J) \models C$.

S *Systematicity*
The collective judgment on each proposition exclusively depends on individual judgments on that proposition. Let $p, q \in \mathcal{S}$ be any propositions and $J = (J_1, \ldots, J_n)$, $J' = (J'_1, \ldots, J'_n)$ be profiles. Then:

$$\textit{if } \forall i : p \in J_i \Leftrightarrow q \in J'_i, \textit{then } p \in f(J) \Leftrightarrow q \in f(J')$$

A *Anonymity*
The voters are treated equally, i.e. a permutation of the individuals does not influence the result:

$$f((J_1, \ldots, J_i, \ldots, J_j, \ldots, J_n)) = f((J_1, \ldots, J_j, \ldots, J_i, \ldots, J_n)) \ \forall J$$

The following theorem was first proved in [List and Pettit, 2002] and embellished in [Pigozzi, 2006].

Theorem 6.1 (List and Pettit's Impossibility Theorem)**.** If $\mathcal{S} \supseteq \{a, b, a \wedge b\}$ (where $\wedge$ could be replaced by $\vee$ or $\supset$), there exists no aggregate procedure that simultaneously satisfies U, R, S and A.

These requirements are explained in our court example. U means that each meaning that satisfies the integrity constraint $V \wedge B \leftrightarrow D$ is a valid input. R states that the final decision also satisfies this condition. S says that the collective acceptance or denial of (V), (B) and (D) only depends on the individual acceptance of nothing but the proposition under consideration. Finally, A makes sure that each juror has the same weight.

The result can even be strengthened by weakening the anonymity to the property of non-dictatorship, [Pauly and Hees, 2006], which states that there is some i s.t.

$$J_i = f(J)$$

Informally, the collective decision is equivalent to the one of some individual. To summarize this informally, List and Pettit's Impossibility Theorem states that a proposition-wise aggregation is impossible, regardless which aggregate function we use [Dietrich and Franz, 2007]. Many related theorems with slightly differing side conditions and conclusions have been proved in recent years.

To strengthen the intuitive understanding of the theorem, we briefly discuss some aggregation procedures following [Dietrich and Franz, 2007]. A simple majority voting rule for each proposition clearly fulfills the property of systematicity but possibly violates collective rationality as we have seen in the motivating example. In contrast the premise-based approach clearly delivers a reasonable group decision but violates systematicity. Dictatorship satisfies both systematicity and rationality (as long as the dictator acts rationally) but is not a reasonable method since it is undemocratic.

Dalal's Operator for Aggregation

We now show that the proposed framework can be used to implement judgment aggregation applications. Because of List and Pettit's Impossibility Theorem, we first need to relax our requirements on the aggregate procedure. Otherwise it is hopeless to find a reasonable function. Following [Pigozzi, 2006], this is done by relaxing systematicity which is controversial anyway, i.e. collective decision about a proposition is not independent from other propositions.

In Section 6.2, we compared premise-based with conclusion-based methods and none of them was satisfactory. Gabriella Pigozzi called her approach *argument-based procedure*. It gives priority

to the satisfaction of the integrity constraints and only minimizes the distance to the individuals secondary. This makes sure that the collective decision is rational and further has some pleasant properties like the possibility to deal with incomplete knowledge.

Pigozzi proposes the use of Dalal's merging operator for computing the final decision, where the distance function is the Hamming distance (as introduced in Section 6.1) and the aggregation function is the sum. Note that the term *aggregation function* is overloaded somehow. First, we use the term in the context of Dalal's operator, where it is a function for combining n distance values into a single one. Second, it is used to name the process of mapping individual decisions onto a collective one.

We now continue the court example. Dalal's operator selects the merged belief base such that its only model is the one among all interpretations satisfying the integrity constraints, which has the minimum overall Hamming distance. Formally:

$$\Delta^n_{h,m,\Sigma}(J, C) = form(\arg \min_{I:I \not\models \bot, I \models C} \Sigma_i h(I, J_i)),$$

where $C = \{V \wedge B \leftrightarrow D\}$.

The set of interpretations which satisfy C is:

$$\{\emptyset, \{V\}, \{B\}, \{V, B, D\}\}$$

Next we need to compute the Hamming distances between each of these interpretations and the individual beliefs.

	$h(I_i, J_1)$	$h(I_i, J_2)$	$h(I_i, J_3)$	$\Sigma_i h(I, J_i)$
$I_1 = \emptyset$	3	1	1	5
$I_2 = \{V\}$	2	2	0	4
$I_3 = \{B\}$	2	0	2	4
$I_4 = \{V, B, D\}$	0	2	2	4

Note that the interpretation $I_c = \{V, B\}$, which exactly corresponds to the invalid group decision in the general part of this section, has a smaller total distance, namely $1 + 1 + 1 = 3$. But it is not considered due to violation of C. Hence the discursive dilemma is avoided.

Unfortunately, the example shows that there are in general more interpretations with minimum distance to the individual decisions. In the example we can only exclude $I_1 = \emptyset$ since it has distance 5, but all other interpretations I_2, I_3 and I_4 have equal distances. Nevertheless Dalal's operator makes a reasonable pre-selection. It excludes (i) interpretations that violate the integrity constraints and (ii) interpretations with non-minimal distance.

A final step is to select one or more of the surviving interpretations. Even though in principal the selection is arbitrary, just randomly selecting one is difficult to justify. The final choice can be driven by the idea of relaxing some of the assumptions. For instance we could drop the principle of anonymity to incorporate different expertise [Pigozzi, 2006]. Some authors like [Eckert and Pigozzi, 2005] just take the disjunction of all outcomes with minimal distance. In our example this means:

$$Mod(\Delta^n_{h,m,\Sigma}(J, C)) = \{\{V\}, \{B\}, \{V, B, D\}\}$$

However, this result is disappointing. Even though we have resolved irrational decisions, we still cannot answer the final question: $D \in J_A$ or $D \notin J_A$? Or in terms of the example: Does the defendant has to pay or not?

Therefore we extend the approach from the literature in the following way. As a final step we apply a majority voting rule. In contrast to the initially explained solutions the majority rule

does *not* select among individual outcomes, but among different *collective decisions*. We define the finally applied selection operator as follows:

Definition 6.6. The unary *majority selection* operator is defined as:

$$\circ^1_{ms} : 2^{\mathcal{A}(\Sigma^C)} \times \Sigma^C_p \to 2^{\mathcal{A}(\Sigma^C)}$$

$$\circ^1_{ms}(SAS, P) = \begin{cases} \{AS \mid AS \in SAS, P \in AS\} & \textit{iff } acc > deny \\ \{AS \mid AS \in SAS, P \notin AS\} & \textit{iff } acc < deny \\ SAS & \textit{otherwise} \end{cases}.$$

where $acc = |\{AS | AS \in SAS, P \in AS\}|$ and $deny = |\{AS | AS \in SAS, P \notin AS\}|$.

The first operator parameter is a set of answer sets and its second one is a propositional atom.

The operator will first check if the majority of all answer sets accept or deny P and then return the set of all answer sets that follow this majority. If the number of accepting and denying answer sets is equal, the operator returns all of them, i.e. it passes the burden of making a final decision to the user, just as the approach in the literature.

Note that the operator is unary. It gets just one set of answer sets, but this single set contains possibly multiple answer sets.

Implementation

The implementation of individual outcomes within the framework proposed in this thesis is straightforward. We just have to encode $J_i \subseteq \mathcal{S}$ as set of facts:

$$P_{J_i} = \{s \mid s \in J_i\}.$$

Then $AS(P_{J_i}) = \{J_i\}$ and is therefore exactly what we expect. If we assume that P_{J_i} is stored in "outcome$i$.hex" it is possible to implement the following merging plan:

```
[common signature]
predicate: v/0; predicate: b/0; predicate: d/0;

[belief base]
name: individual1;
source: "outcome1.hex"

...

[belief base]
name: individualN;
source: "outcomeN.hex"

[merging plan]
{
        operator: majorityselection;
        majorityOf: "d";
        source: {
                operator: dalal;
                constraint: "← v, b, not d.";
                constraint: "← not v, b, d.";
                constraint: "← v, not b, d.";
                constraint: "← not v, not b, d.";
                source: {individual1};
```

Name (key)	Domain	Descriptions
constraint	dlvhex syntax	Specifies constraints that need to be satisfied by the group decision
constraintfile	local paths and filenames	Specifies files containing dlvhex programs that need to be satisfied by the group decision
aggregation	one of "max" or "sum"	Selects the aggregation function D (see Section 6.1); default is "sum"
penalize	one of "ignoring", "unfounded" or "aberration"	Defines the distance function $d(AS_I, AS_A)$ (see Section 6.1), where AS_I is the belief set of an individual and AS_A the group belief set. "ignoring" will penalize situations where $a \in AS_I \wedge a \notin AS_A$ or $\neg a \in AS_I \wedge \neg a \notin AS_A$ for some proposition a. "unfounded" will penalize situations where $a \in AS_A \wedge a \notin AS_I$ for some proposition a. "aberration" penalizes both "ignoring" and "unfounded".
weights	$\mathbb{R}^n$ where n is the number of jurors	Sets the weights of the jurors. Higher values denote greater influence. Default is equal weight for all jurors.

Table 6.1: Parameters of Dalal's operator

```
                ...
                source: {individualN};
        };
}
```

(Note: The switch from capital to lower case letters was necessary because upper case letters denote variables in dlvhex.)

We first apply the minimum Hamming distance operator in order to find all collective decisions that respect the integrity constraints and have minimum distance to the individuals. The result is passed to the majority selection operator, which will make a final decision if possible.
In case of our running example, i.e,

$$AS(P_{J_1}) = \{\{v, b, d\}\}, AS(P_{J_2}) = \{\{v, \neg b, \neg d\}\}, AS(P_{J_3}) = \{\{\neg v, b, \neg d\}\}$$

the final result of the merged belief base will be

$$AS(P_A) = \{\{v, \neg b, \neg d\}, \{\neg v, b, \neg d\}\}.$$

Thus, it is fixed that d is denied (i.e., the defendant does not have to pay), where this is justified either by $\neg b$ or $\neg v$, depending on the individual we ask. Without the final majority selection, also $\{v, b, d\}$ and therefore the acceptance of d would be a possible group outcome because it has the same Hamming distance to the individual votes.

Operator Parameters

The previous example shows the simplest usage of the operator. Several optional parameters were implemented that allow customization. Table 6.1 summarizes them.

We demonstrate the usage of parameter "penalize".

```
[common signature]
predicate: a/0; predicate: b/0;

[belief base]
name: individual1;
mapping: "a v b."

[belief base]
name: individual2;
mapping: "a."

[belief base]
name: individual3;
mapping: "b."


[merging plan]
{
        operator: dalal;
        penalize: ignoring;
        source: {individual1};
        source: {individual2};
        source: {individual3};
}
```

Obviously $AS(P_{J_1}) = \{\{a\}, \{b\}\}$, $AS(P_{J_2}) = \{\{a\}\}$, $AS(P_{J_3}) = \{\{b\}\}$. If we penalize only "ignoring", i.e. individual beliefs that are not in the group decision, the only aggregated judgment with minimal cost is $\{a, b\}$ which has distance 0 to all three jurors, even though it contains *more* than each of the individual beliefs (but this is not penalized).

In contrast, if we penalize both "ignoring" and "unfounded", the aggregated belief set will be alternatively $\{a\}$ or $\{b\}$ because they match two of the jurors exactly (cost 0) and differ from the third one in two literals. $\{a, b\}$ is *not* a group decision with minimal cost because it differs from *each* individual in 2 literals causing cost $3 \cdot 2 = 6 > 2$.

Other Logics

In the previous sections we considered propositions in the sense of *propositional logic*. Nevertheless similar principals can be applied for other similar logics [Dietrich and Franz, 2007].

Since dlvhex perfectly supports predicates, at least first-order statements can easily included in the proposed framework.

Judgment Scenarios

Apart from court cases, other examples for group decision making can be found in many practically relevant domains: [Everaere et al., 1999, Delgrande and Schaub, 2007, Dietrich and Franz, 2007]

- Planning group activities with individual preferences, e.g., holiday planning;
- Selecting candidates by juries, e.g., granting scholarships to the best qualified students, where *best qualified* is measured on several attributes like publications, marks and social skills.
- Merging signals from several sensors in robotic applications.
- Political debates and polls, e.g., parties independently decide which of three propositions they accept

(B) The birth rate is too low

(C) If the birth rate is too low then more immigration is needed

(I) More immigration is needed
with the inherent rationality criterion $B \wedge C \leftrightarrow I$.

- Merging expert opinions concerning fault diagnosis

This shows that the task of judgment aggregation can elegantly be solved using the proposed framework. However, this can also be done by hand or using existing systems. The actual benefit when the mergingplugin is used, is that the strategy can be modified very quickly. For instance, if we decide that the final majority selection should be removed, such that all three group decisions with minimal distance survive, we simply need to remove the outermost operator application from the merging plan file. Further, if we decide to use an operator different from Dalal's, this can be done by editing just one token in the task definition. Then the framework will automatically create a HEX program implementing the changed semantics.

6.3 Fault Diagnosis

In the last section we have shown that the framework can be used to merge judgments from several individuals. Now we extend this idea further and consider fault diagnosis. Informally speaking, fault diagnosis deals with the finding of explanations for observed (fault) behavior. Formally we can define the task as follows [Eiter and Gottlob, 1995].

Definition 6.7. A propositional abduction problem (PAP) consists of a tuple $\mathfrak{P} = \langle V, H, M, T \rangle$, where V is a finite set of propositional variables, $H \subseteq V$ is a set of hypothesis, $M \subseteq V$ is the set of manifestations and T is a consistent propositional theory over V.

Definition 6.8. Let $\mathfrak{P} = \langle V, H, M, T \rangle$ be a propositional abduction problem. A set $S \subseteq H$ is a solution iff $T \cup S$ is consistent and $T \cup S \models M$.

Full Adder Basic Definition

As our running example in this section, we are going to consider the circuit diagram of a full adder, see Figure 6.1. The full adder *fa* consists of two half adders *ha1* and *ha2* as well as a final *or*-gate *faOr*. Each of the half adders contains one *and*-gate *haAnd1* resp. *haAnd2* and one *xor*-gate *haXor1* resp. *haXor2*.

We are going to model the system "fulladder.dl" such that we can set its input by defining facts of kind $in(I, component, V)$ where $I \in \{x, y, c\}$ denotes the input line (with c being the carry flag c_{in}) and $V \in \{0, 1\}$ the truth value on this wire. Similarly the output is encoded in atoms of form $out(I, component, V)$ with $I \in \{s, c\}$ with s being the output signal and c the carry (c_{out}).

The full modeling is given in Appendix C. Basically it consists of the component definitions and the wiring between them. As an example, the definition of and *and*-gate is as follows.

$$
\begin{aligned}
P_{and} = \{ & and(haAnd2). \\
& out(s, A, 0) \leftarrow and(A), in(x, A, 0). \\
& out(s, A, 0) \leftarrow and(A), in(y, A, 0). \\
& out(s, A, 1) \leftarrow and(A), in(x, A, 1), in(y, A, 1).\}
\end{aligned}
$$

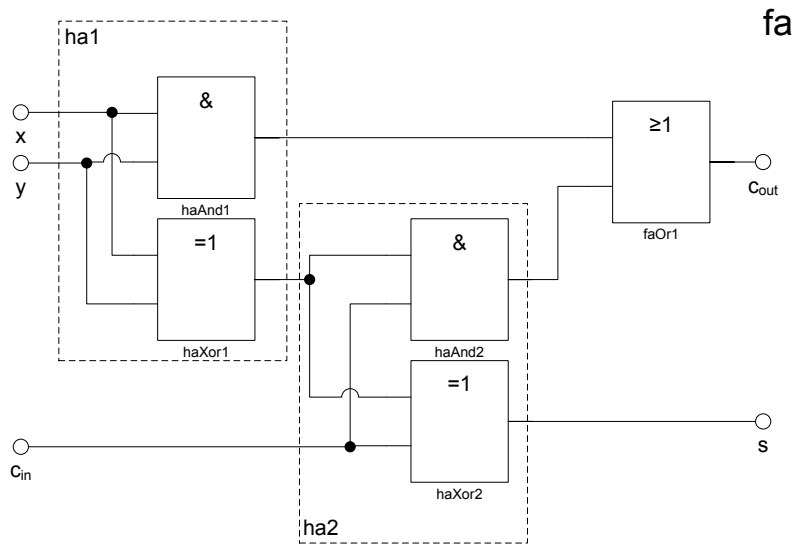


Figure 6.1: Full adder modeling

Example 6.3. Suppose we add facts $in(x, component, 1)$, $in(y, component, 1)$ and $in(c, component, 0)$. Then the reasoner will derive $out(s, component, 0)$ and $out(c, component, 1)$.

Modeling Malfunction

In order to define a reasonable abduction problem, we now modify our modeling such that the components deliver the expected output for sure only if they work normally. Otherwise they may return false results. This is modeled by suspending the according rules in case of a fault in component X, denoted as $ab(X)$. For instance, the refined modeling of the *and*-gate is as follows:

$$
\begin{aligned}
P'_{and} = \{ & and(haAnd2). \\
& out(s, A, 0) \leftarrow and(A), \text{not } ab(A), in(x, A, 0). \\
& out(s, A, 0) \leftarrow and(A), \text{not } ab(A), in(y, A, 0). \\
& out(s, A, 1) \leftarrow and(A), \text{not } ab(A), in(x, A, 1), in(y, A, 1).\}
\end{aligned}
$$

Abnormal behavior can be detected as follows. The user does not only specify the input but also the observed output. For this purpose we introduce a predicate

$$outObserved(I, component, V)$$

to denote that we observe that output wire I has value V. Analogously we use *inObserverd* for

input wires. Then we can write the following constraints to detect malfunctioning.

$$\begin{aligned} P = \{ \quad & \leftarrow \quad outObserved(N, C, V), \text{not } out(N, C, V). \\ & \leftarrow \quad inObserverd(N, C, V), \text{not } in(N, C, V). \\ & \leftarrow \quad out(N, C, V1), outObserved(N, C, V2), V1 != V2. \\ & \leftarrow \quad in(N, C, V1), inObserverd(N, C, V2), V1 != V2.\} \end{aligned}$$

The first two lines make sure that all observed output values are actually created by the system. The latter ones guarantee that *only* those values are derived.

Solving the Abduction Problem

In order to solve the abduction problem, we need to define our hypothesis. Basically, all five gates are potentially defect whereas the wires in between are assumed to work correctly in any case, i.e.:

$$P_{hyp} = \{ab(faOr).ab(haXor1).ab(haAnd1).ab(haXor2).ab(haAnd2).ab(faXor).\}$$

Now we can encode arbitrary observations as sets of facts. Let's consider the interpretation that assigns 1 to all input wires. Then in case of correct functioning of the system, the output values and intermediate results are as shown in Figure 6.2.

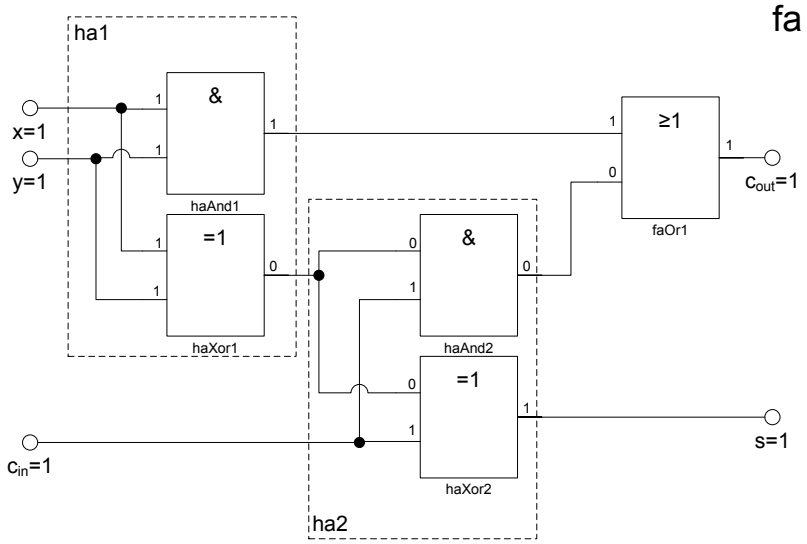


Figure 6.2: Full adder in a faultless scenario

Now we assume that there is a fault in the system that leads to the observation $s = 0$. The carry flag is still correctly computed. Fomally we can describe this setting as follows.

$$
\begin{aligned}
P_{obs} = \{&in(x, component, 1).\\
&in(y, component, 1).\\
&in(c, component, 1).\\
&outObserved(s, component, 0).\\
&outObserved(c, component, 1).\}
\end{aligned}
$$

Clearly the fault must be somewhere between the erroneous output and the input values. In Figure 6.3 this critical path is printed bold and contains the two *xor*-gates. Therefore a reasonable minimal explanation will be the malfunctioning of one of them.

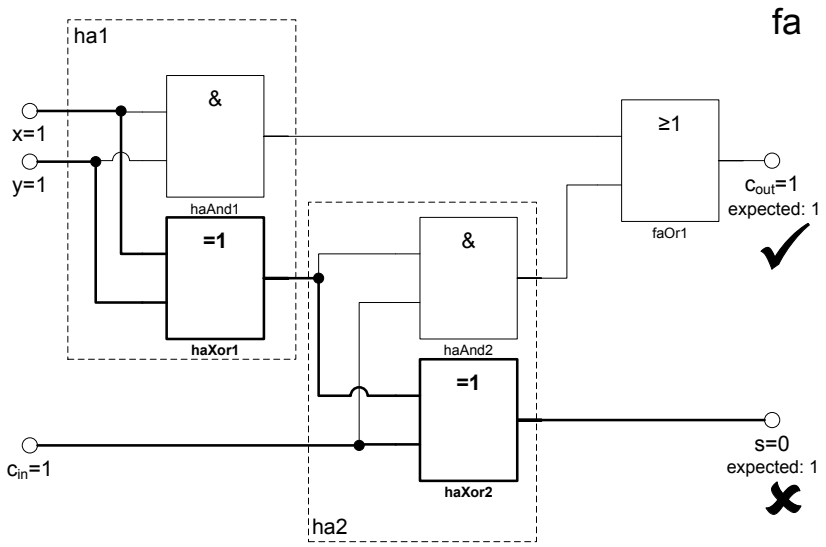


Figure 6.3: Full adder malfunctioning

This intuitive consideration perfectly coincides with the output of DLV when run for this abduction problem. We start the reasoner with the command

```
dlv -FRmin fulladder.dl fulladder.hyp fulladder.obs
```

and retrieve the explainations $\{ab(haXor1)\}$ and $\{ab(haXor2)\}$.

Diagnosis Aggregation

Suppose that we have two or more different explanations for a certain observation. This can be the case if several experts are involved, each with different opinions and expertise.

Continuing our example, assume we have three engineers in total. The first one has the set of hypotheses introduced above. The second one however is less experienced and therefore overlooks that the *xor*-gate in half adder 1 can cause the fault. Similarly, the third expert misses the possibility for a fault in half adder 2. Consequently jurors 2 and 3 think they know for sure the component that is defect, whereas the more advanced expert 1 sees that there are several possibilities. In terms of answer sets this means $AS(P_{J_1}) = \{\{ab(haXor1)\}, \{ab(haXor2)\}\}$, $AS(P_{J_2}) = \{\{ab(haXor2)\}\}$, and $AS(P_{J_3}) = \{\{ab(haXor1)\}\}$.

We begin to model our sources in the merging plan file as follows:

```
[common signature]
predicate: ab/1;

[belief base]
name:juror1;
dlvargs: "-FRmin fulladder.dl abnormal1.hyp fault.obs";

[belief base]
name: individual2;
dlvargs: "-FRmin fulladder.dl abnormal2.hyp fault.obs";

[belief base]
name: individual3;
dlvargs: "-FRmin fulladder.dl abnormal2.hyp fault.obs";
```

This will call DLV in the background and solve the three abduction problems, each with different hypotheses. Alternatively to the call of an abductive reasoner, we could also hard-code the individual beliefs. Now we come to the point where the three explanations need to be merged. This is done using Dalal's operator. In terms of our merging task this looks as follows.

```
[merging plan]
{
        operator: dalal;
        constraintfile: "fulladder.dl";
        constraintfile: "fault.obs";
        {juror1};
        {juror2};
        {juror3};
}
```

Note that we need to add the complete definition of the full adder and the observation again as constraint source, because the final group decision must be a valid explanation for the fault. If we would not respect that, we would get an aggregated decision that has minimal distance to the expert's, but possibly does not explain our observation. This is similar to the court case example where a simple majority voting rule leads to irrational results (see 6.2).

The result of the above merging task is $E_1 = \{ab(haXor1), ab(haXor2)\}$, i.e., the experts agree that *both* components are defect. Clearly, this is not a minimal explanation anymore, but it is still valid and has minimal distance to the individuals according to our cost model because it respects the opinions of all experts, i.e. there are no individual beliefs that are not contained in the aggregated decision. This may be undesired result is caused by the kind we compute distances. E_1 contains all of the individual's beliefs but, from the viewpoint of one of the experts, *even more* atoms. However, we do not penalize atoms in the group decision that are not in an individual's but only the other way round. Table 6.2 summarizes the results.

We now modify our setting such that not only ignoring individual beliefs is penalized, but also additional propositions that are not founded for some expert. In this case the group explanations are $E_2 = \{ab(haXor1)\}$ and $E_3 = \{ab(haXor2)\}$. Both have distance 0 to expert 1, and distances 2/0 resp. 0/2 to experts 2/3[1] For instance, $E_3 = \{ab(haXor2)\}$ has distance 2 to expert 3, because it ignores the expert's belief that *haXor1* is defect and adds some other proposition (*haXor2* is malfunctioning), which is not founded for expert 3.

In contrast $\{ab(haXor1), ab(haXor2)\}$ is not an explanation with minimal distance anymore because its distance to each expert is 1 and therefore has cost 3 in total.

[1]Remember that expert 1 delivers several answer sets that are considered as *alternatives*. i.e., we always compare with the *best-fitting* answer set, see Section 6.1.

	$AS(P_{J_1})$	$AS(P_{J_2})$	$AS(P_{J_3})$	Sum
$\mathbf{E_1}$ with penalizing only "ignoring"	0	0	0	**0** $\Leftarrow$
E_2 with penalizing only "ignoring"	0	1	0	1
E_3 with penalizing only "ignoring"	0	0	1	1
E_1 with penalizing every "aberration"	1	1	1	3
$\mathbf{E_2}$ with penalizing every "aberration"	0	2	0	**2** $\Leftarrow$
$\mathbf{E_3}$ with penalizing every "aberration"	0	0	2	**2** $\Leftarrow$
E_1 with weights w, penalizing every "aberration"	1	2	1	5
E_2 with weights w, penalizing every "aberration"	0	4	0	4
$\mathbf{E_3}$ with weights w, penalizing every "aberration"	0	0	2	**2** $\Leftarrow$

where

$E_1 = \{ab(haXor1), ab(haXor2)\}$
$E_2 = \{ab(haXor1)\}$
$E_3 = \{ab(haXor2)\}$
$AS(P_{J_1}) = \{\{ab(haXor1)\}, \{ab(haXor2)\}\}$
$AS(P_{J_2}) = \{\{ab(haXor2)\}\}$
$AS(P_{J_3}) = \{\{ab(haXor1)\}\}$

and the weights are $w = (1, 2, 1)$

Table 6.2: Distances of explanations E_i to the experts

Now assume that it turns out that expert 2 has not overseen the possibility that *haXor1* could be defect (he is less experienced, as we said), but he has successfully tested the component and had therefore excellent reasons for excluding it in the diagnosis task. We can model this by adding weights to the experts. Suppose our weight vector is $w = (1, 2, 1)$, i.e., expert 2 is two times as important as the others. This gives us the only remaining explanation $E_3 = \{ab(haXor2)\}$ since the distances to the experts have changed, as depicted in Table 6.2. Whereas E_2 and E_3 have equal summed distance with equal weights, E_2 has now greater costs because it is inconsistent with the (more important) expert 2.

This example demonstrates the strengths of the framework. As we have seen it is possible to experiment with different distance measurements without coding them from scratch. It was necessary to implement Dalal's operator only once. Now it can be parametrized and used in a variety of applications.

Popular aggregation functions different from sum include for instance average and maximum. Computing the maximum results in a *minimax* optimization criterion, i.e., the maximum of all distances to the n belief bases is minimized.

Extensions and Outlook

The scenario can be extended in several ways. One interesting kind is to neglect the assumption that each expert acts rationally but still demanding the group decision to be rational.

For instance, assume we have the following propositions [Dietrich and Franz, 2007]:

(B) The birth rate is too low

(C) If the birth rate is too low, then more immigration is needed

(I) More immigration is needed

Then an obvious rationality condition is $B \wedge C \leftrightarrow I$ (or at least $B \wedge C \rightarrow I$). Nevertheless we could imagine some individual opinion, for instance a political party, that accepts both B and

C but denies I. This kind of irrational behavior is perfectly supported by the proposed operator implementation, because it primarily selects the final decision among all *valid* interpretations and minimizes the distance to individuals only secondarily.

6.4 Merging of Relational Data

Even though professional database systems should be preferred when working with relational data since they are optimized for high-performance throughput even in case of large amounts of data, we briefly discuss a task from this field to show that in principal the framework is generalizable. *Schema mapping* is a popular research topic which deals with the incorporation of differing data models. In contrast, *data merging* works at the level of data tuples and has been neglected until a few years ago [Naumann and Häussler, 2002].

A closer look at this task reveals two steps. The first one is the identification of syntactically different entries that refer to the same real-world object. The second one is the resolution of this conflict. If we assume that the first step was already performed, we have a unique ID for each object and the remaining task is essentially the implementation of an advanced join operation. The most reasonable kind will probably transfer attributes, upon which all sources agree, directly into the merged table. Attributes that occur only in one table are filled up with default values in the tuples from the other one (i.e., we perform a full outer join). For all other attributes, i.e. attributes that contradict each other, an aggregation function needs to be defined, e.g., using an *unknown* value, applying a majority rule or computing the average (in case of numeric values).

If we express the source data to be merged using predicates with an arity that matches the number of columns of the according table, we can easily express relational tables as logic programs. Then the framework is applicable and we could implement usual join operations, or we can use it to repair inconsistencies in databases, though it is stressed again that this is only useful for making experiments rather than for merging real-world databases.

6.5 Further Scenarios and References

Further scenarios for the belief merging framework include ontologies and biomedical applications. Ontologies summarize the terms and relations between terms that are relevant for a certain domain. Merging of several ontologies is a special challenge that arises in Semantic Web applications due to decentralization [Stumme and Maedche, 2001].

In medical diagnosis, decision trees are an important aid. For instance, in case-based reasoning they are used in order to find protocoled cases that are similar to the current one. This will reveal which of the therapy choices are promising and which are not. For an elaborated discussion of this topic, we refer to [Redl, 2010].

We can only see a short distance
ahead, but we can see plenty
there that needs to be done.
Alan Turing

Chapter 7

Conclusion and Outlook

7.1 Problem Statement

In some applications we work with multiple data sources. If they are provided by third parties, they are rarely synchronized. Then it is necessary to incorporate them into one, in order to get a consistent single point of truth. Clearly, the merging is not a trivial task and it is often the case that we don't know right from the beginning which strategy is the best. Traditionally, the user has the burden to implement several strategies and make experiments to determine the best suited one. But this task also contains a lot of routine work.

It is therefore desirable to support this task with a tool that manages technical details, such that the user can focus on the development and optimization of the merging algorithms.

7.2 Solution

The framework developed in this thesis minimizes routine work by hiding technical details behind a user-friendly merging plan language, where the information flow between different merging strategies, called *merging operators*, can be specified declaratively. The merging language allows the definition of a *merging plan*, which is a tree-like structure with the belief bases in the leaf nodes and merging operators in the inner nodes. This is very similar to arithmetic expressions. The final result of a merging plan is the return value of the topmost merging operator.

The framework comes with a few preinstalled merging operators, but can easily be extended with user-defines ones by implementing a standardized interface. This ensures, that the framework is flexible enough to use it in many different situations. The actual benefit of the framework becomes clear when it is not known right from the beginning which merging strategy will behave best. Then the user can try out several merging plans and evaluate the results empirically. This allows him to focus on the development and optimization of the conflict resolution techniques without dealing with technical details. If the operators in use shall be exchanged or parameterized in a new way, this requires only minimal changes in the declarative task definition since the framework will do the rest of the job, i.e., the recomputation of the result according to the new merging plan.

Finally we have demonstrated the framework in Section 6.3 using a case study about aggregating multiple experts decisions in fault diagnosis tasks. For this purpose, we solved several propositional abduction problems, one for each agent, and finally combined the individual's results using a Dalal-style operator as implemented for the framework.

In order to simplify the translation of merging plans into HEX programs (Section 5.7) the concept of *nested* HEX *programs* was developed and implemented as part of this thesis (Chapter 3). This feature can also be used independently from the merging framework, for instance to outsource parts of the logical program and partition a program into several modules like in classical procedural programming. This allows a team of developers to work simultaneously on different parts of the program.

More generally, this feature allows the user to reason on the level of sets of answer sets. This supports us if we need to know the contents of *all* answer sets of a (sub-)program for computing some result. A typical example is the answering of a query under cautious reasoning, which may be not directly supported by the reasoner.

By June 2010, only very few belief merging frameworks were actually available in form of implemented systems. The tool developed in this thesis is definitely the most general one due to its generic design. It does not make any assumptions about the merging task, but can be used for arbitrary scenarios, as long as suitable operators are provided. This ensures a maximum of flexibility.

7.3 Future Issues

The merging language still leaves much room for enhancements. Currently it supports the definition of a common signature and mapping rules that solve the problem of syntactic incompatibility, i.e., the usage of different knowledge representation formalisms in the sources. It further allows the specification of the information flow through the merging operators in form of a hierarchical merging plan.

At the moment the language is thought as a kind of sandbox for testing conflict involvement strategies. In order to be applicable also in real world scenarios, it is necessary to pay special attention to performance. This can be accounted for by merging plan *optimization* where plans are reorganized into semantically equivalent but computationally advantageous variants. This is similar query optimization as performed by database systems (see for instance [Chaudhuri, 1998]). The first step is to enhance the language such that properties like associativity and commutativity, which are fulfilled by the operators, can be specified. This allows the merging plan compiler to determine the equivalence of revision plans. In a further step we need to define a cost model to find the fastest of several semantically equal merging plans. Such a model can, for instance, implement the idea of minimizing the size of intermediate results.

Another possibility for future enhancements is the extension of the repertoire of operators. Currently there are implemented a few trivial operators (like set union), and an operator defined on top of Dalal's distance function (Chapter 6). These operators can either be extended by introducing additional parameters, and completely new merging operators can be added, for instance variants of the revision operators developed by Winslett [1988] and Ginsberg [1986]. A larger set of predefined operators enhances the chances that the user will find suitable operators for a concrete merging task and therefore gets rid of the burden of implementing them himself.

The management of the answers of nested HEX programs operators also has room for improvement. As we discussed in Section 5.1 and visualized in Figure 5.2, the answers are written into an internal cache. Currently the answers resist there until the dlvhex instance terminates. This is reasonable for small applications, but we can run out of memory in case of huge knowledge sources. Therefore, typical caching strategies could be implemented that remove old cache

entries if they have not been accessed for a certain time. Clearly, this has the disadvantage that the program that delivered the answer needs to be executed again if its answer is accessed later on. In other words, we pay memory friendly caching strategies with possibly longer runtime.

Finally, a possible far-reaching extension is the merging of programs rather than the merging of answer sets. This enables the user to keep the constraints defined in the individual sources. Currently this is only possible indirectly by redefining them for the merged base, see for instance the examples in Section 6.1.

Appendix A

Merging Plan Language

The set of all syntactically correct merging plans $\mathcal{R}$ is given by the grammar in Table A.1. It incorporates all requirements from Chapter 5.

Terminal tokens are underlined. Note that the scanner of the implemented merging plan compiler skips whitespaces (tabs, blanks, newlines) between tokens automatically. This is not shown in the grammar in order to keep it as simple as possible.

Lexer		
Literal	⇒	$\underline{-}?\ \Sigma_p\ (((\Sigma_c\|\Sigma_v)(\underline{,}\ \Sigma_c\|\Sigma_v)^*))?$
PredicateName	⇒	$[\underline{a}-\underline{z}]\ ([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}]\|[\underline{0}-\underline{9}])^*$
KBName	⇒	$([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}])\ ([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}]\|[\underline{0}-\underline{9}])^*$
OPName	⇒	$([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}])\ ([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}]\|[\underline{0}-\underline{9}])^*$
Variable	⇒	$([\underline{A}-\underline{Z}])\ ([\underline{a}-\underline{z}]\|[\underline{A}-\underline{Z}]\|[\underline{0}-\underline{9}])^*$
Number	⇒	$([\underline{1}-\underline{9}][\underline{0}-\underline{9}]^*)\ \|\ \underline{0}$
General ASP Grammer		
Fact	⇒	*RuleHead* .
Constraint	⇒	: − *RuleBody* .
Query	⇒	not? *Literal* (, not? *Literal*)*
RuleHead	⇒	*Literal* (∨ *Literal*)*
RuleBody	⇒	*Query*
Rule	⇒	*RuleHead* : − *RuleBody* .|*Fact*|*Constraint*
Merging Plan Specific Grammer		
Program	⇒	*CommonSigDef* *Mappings* *MergingPlan*
CommonSigDef	⇒	[*common signature*] *PredicateDefinition**
Mappings	⇒	*KnowledgeBase**
MergingPlan	⇒	[*merging plan*] *MergingPlanNode*
MergingPlanNode	⇒	{ *operator* : *OPName* ; (*key* : *value* ;)* (*source* : *MergingPlanNode* ;)* } | *KBName*
PredicateDefinition	⇒	*predicate* : *PredicateName* / *Number* ;
KnowledgeBase	⇒	[*knowledge base*] *name* : *KBName* ; (*MappingRule**)|*ExternalSource*
MappingRule	⇒	*mapping* : "*Rule*";
ExternalSource	⇒	*source* : *Filename*;
stringliteral	⇒	" $\{"\}^{c*}$ " (where S^c is the complement of set S)
Filename	⇒	*stringliteral*
key	⇒	Σ_c|*stringliteral*
value	⇒	Σ_c|*stringliteral*

Table A.1: Syntax of merging compiler input files

Appendix *B*

Merging Plan Compiler

B.1 Options

The mpcompiler recognizes the following command-line options:

- `-parsetree`
  Generates a parse tree rather than dlvhex code (mostly for debug purposes).
- `-help`
  Prints an online help message.
- `-spirit` or `-bison`
  Forces the compiler to use a *boost spirit* resp. *bison* generated parser. Default is spirit.

If no filenames are passed, the compiler will read from standard input. If at least filename is passed, standard input will *not* be processed by default. However, if `--` is passed as additional parameter, standard input will be read additionally to the input files.

B.2 Merging Plan Files

The merging scenario is defined in merging plan files of the following form:

```
[common signature]
predicate: pred1/arity1;
...
predicate: predN/arityN;

[belief base]
name: nameOfBeliefBase1;
mapping: "head1 :- body1."
...
mapping: "headM :- bodyM."

...

[belief base]
name: nameOfBeliefBaseK;
```

```
mapping: "head1 :- body1."
...
mapping: "headJ :- bodyJ."

[merging plan]
{
operator: someOperatorsName;
key1: value1;
...
keyN: valueN;
source: {
        operator: subPlanOperator;
        ...
        source: {nameOfBeliefBase1};
        source: {nameOfBeliefBase2};
};
source: {
        ...
};
}
```

Essentially the file consists of 3 sections.

Common Signature

In statements of form

predicate: pred1/arity1;

all relevant predicates that occur in the belief bases are defined. Those predicates will be output by dlvhex after the merging plan was processed.

Belief Bases

Belief bases can be any data source: relational databases, XML files, etc.. The only requirement is that they are accessible from dlvhex through an appropriate external atom. Belief bases are defined by blocks of form:

```
[belief base]
name: nameOfBeliefBase1;
mapping: "head1 :- body1.";
...
mapping: "headM :- bodyM.";
```

where the name defines a legal name for this belief base, followed by an arbitrary number of *mappings*. Mappings can essentially be arbitrary dlvhex code fragments. However, in reasonable applications they access the underlying (proprietary) belief base and map their content onto the common signature (see above).

Alternatively they can also be defined by

```
[belief base]
name: nameOfBeliefBase1;
source: "externalfile.hex";
```

where “externalfile.hex" is an external file containing (computation source access rules and) mapping rules. Note that *mapping* and *source* cannot be used simultaneously.

Merging Plan

The merging plan is a hierarchical structure that combines the belief bases such that only one final result survives at the end of the day. A merging plan section is of form:

```
operator: XYZ.
key1: value1;
...
keyN: valueN;
source: ...;
source: ...;
```

Such a section defines the operator to apply, the key-value pairs that shall be passed to the operator and the sub merging plans (source). A sub merging plan (after a source statement) can either be a belief base (denoted as {bbName};) or a *composed merging* plan (i.e., the result of a prior operator application).

A more formal definition of the merging file format is given in Appendix A.

Appendix *C*

Full Adder

The modeling of our full adder from Section 6.3 is as given in Example C.1.

The complete set of hypothesis is shown in Example C.2. Note that some of the jurors in Section 6.3 use only parts of this set.

Example C.1.

$$
\begin{aligned}
P_{fulladder} = \{ \quad & or(faOr).or(haXor1).and(haAnd1).or(haXor2).and(haAnd2). \\
& halfadder(HA1, haAnd1, haXor1).halfadder(HA2, haAnd2, haXor2). \\
& fulladder(fa1, HA1, HA2, faOr). \\
\%system & \\
in(x, fa1, V) \leftarrow \quad & in(x, system, V). \\
in(y, fa1, V) \leftarrow \quad & in(y, system, V). \\
in(c, fa1, V) \leftarrow \quad & in(c, system, V). \\
out(x, system, V) \leftarrow \quad & out(x, fa1, V). \\
out(y, system, V) \leftarrow \quad & out(y, fa1, V). \\
\%constraints & \\
\leftarrow \quad & outObserverd(N, C, V), \text{not } out(N, C, V). \\
\leftarrow \quad & inObserverd(N, C, V), \text{not } in(N, C, V). \\
\leftarrow \quad & out(N, C, V1), outObserverd(N, C, V2), V1 != V2. \\
\leftarrow \quad & in(N, C, V1), inObserverd(N, C, V2), V1 != V2. \\
\%or & \\
out(x, O, 1) \leftarrow \quad & or(O), \text{not } ab(O), in(x, O, 1). \\
out(x, O, 1) \leftarrow \quad & or(O), \text{not } ab(O), in(y, O, 1). \\
out(x, O, 0) \leftarrow \quad & or(O), \text{not } ab(O), in(x, O, 0), in(y, O, 0). \\
\%and & \\
out(x, A, 0) \leftarrow \quad & and(A), \text{not } ab(A), in(x, A, 0). \\
out(x, A, 0) \leftarrow \quad & and(A), \text{not } ab(A), in(y, A, 0). \\
out(x, A, 1) \leftarrow \quad & and(A), \text{not } ab(A), in(x, A, 1), in(y, A, 1). \\
\%xor & \\
out(x, X, 1) \leftarrow \quad & xor(X), \text{not } ab(X), in(x, X, V1), in(y, X, V2), V1 != V2. \\
out(x, X, 0) \leftarrow \quad & xor(X), \text{not } ab(X), in(x, X, V), in(y, X, V). \\
\%halfadder & \\
in(x, HAnd, V) \leftarrow \quad & halfadder(H, HAnd, HXor), \text{not } ab(H), in(x, H, V). \\
in(y, HAnd, V) \leftarrow \quad & halfadder(H, HAnd, HXor), \text{not } ab(H), in(y, H, V). \\
in(x, HXor, V) \leftarrow \quad & halfadder(H, HAnd, HXor), \text{not } ab(H), in(x, H, V). \\
in(y, HXor, V) \leftarrow \quad & halfadder(H, HAnd, HXor), \text{not } ab(H), in(y, H, V). \\
out(x, H, V) \leftarrow \quad & halfadder(H, HAnd, HXor), \text{not } ab(H), out(x, HAnd, V). \\
out(y, H, V) \leftarrow \quad & halfadder(H, HAnd, HXor), \text{not } ab(H), out(x, HXor, V). \\
\%fulladder & \\
in(x, HA1, V) \leftarrow \quad & fulladder(F, HA1, HA2, For), \text{not } ab(F), in(x, F, V). \\
in(y, HA1, V) \leftarrow \quad & fulladder(F, HA1, HA2, For), \text{not } ab(F), in(y, F, V). \\
in(x, HA2, V) \leftarrow \quad & fulladder(F, HA1, HA2, For), \text{not } ab(F), out(y, HA1, V). \\
in(y, HA2, V) \leftarrow \quad & fulladder(F, HA1, HA2, For), \text{not } ab(F), in(c, F, V). \\
in(x, For, V) \leftarrow \quad & fulladder(F, HA1, HA2, For), \text{not } ab(F), out(x, HA1, V). \\
in(y, For, V) \leftarrow \quad & fulladder(F, HA1, HA2, For), \text{not } ab(F), out(x, HA2, V). \\
out(x, F, V) \leftarrow \quad & fulladder(F, HA1, HA2, For), \text{not } ab(F), out(y, HA2, V). \\
out(y, F, V) \leftarrow \quad & fulladder(F, HA1, HA2, For), \text{not } ab(F), out(x, For, V).\}
\end{aligned}
$$

Example C.2.

$$P_{hyp} = \{ab(faOr).ab(haXor1).ab(haAnd1).ab(haXor2).ab(haAnd2).ab(faXor).\}$$

Bibliography

[Alchourròn et al., 1985] Alchourròn, C. E., Gärdenfors, P., and Makinson, D. (1985). On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50:510–530. URL: http://www.jstor.org/pss/2274239.

[Alon et al., 2003] Alon, N., Milo, T., Neven, F., Suciu, D., and Vianu, V. (2003). Typechecking xml views of relational databases. *ACM Trans. Comput. Logic*, 4(3):315–354. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.4111.

[Apt and Bol, 1994] Apt, K. R. and Bol, R. (1994). Logic programming and negation: A survey. *JOURNAL OF LOGIC PROGRAMMING*, 19:9–71. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.20.7217.

[Baader and Snyder, 1999] Baader, F. and Snyder, W. (1999). Unification theory.

[Bairoch and Apweiler, 1997] Bairoch, A. and Apweiler, R. (1997). The swiss-prot protein sequence data bank and its supplement trembl. *Nucleic acids research*, 25(1):31–36. URL: http://view.ncbi.nlm.nih.gov/pubmed/9016499.

[Chaudhuri, 1998] Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43, New York, NY, USA. ACM. URL: Anoverviewofqueryoptimizationinrelationalsystems.

[Dalal, 1988] Dalal, M. (1988). Updates in propositional databases. *Technical Report DCS-TR-222*.

[Decker et al., 2000] Decker, S., Melnik, S., van Harmelen, F. v., Fensel, D., Klein, M. C. A., Broekstra, J., Erdmann, M., and Horrocks, I. (2000). The semantic web: The roles of xml and rdf. *IEEE Internet Computing*, 4(5):63–74. URL: http://citeseer.ist.psu.edu/decker00semantic.html.

[Delgrande and Schaub, 2007] Delgrande, J. P. and Schaub, T. (2007). Two approaches to merging knowledge bases. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.3316.

[Dietrich and Franz, 2007] Dietrich and Franz (2007). A generalised model of judgment aggregation. *Social Choice and Welfare*, 28(4):529–565.

[Dietrich and Mongin, 2010] Dietrich, F. and Mongin, P. (2010). The premiss-based approach to judgment aggregation. *Journal of Economic Theory*. URL: http://dx.doi.org/10.1016/j.jet.2010.01.011.

[Eckert and Pigozzi, 2005] Eckert, D. and Pigozzi, G. (2005). Belief merging, judgment aggregation, and some links with social choice theory. In *In Belief Change in Rational Agents: Perspectives from Artificial Intelligence, Philosophy, and Economics, Dagstuhl Seminar Proceedings 05321*. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.3428.

[Eiter and Gottlob, 1995] Eiter, T. and Gottlob, G. (1995). The complexity of logic-based abduction. *J. ACM*, 42(1):3–42.

[Eiter et al., 2005] Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2005). A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05*, pages 90–96. Professional Book. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.8944.

[Eiter et al., 2006] Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2006). dlvhex: A system for integrating multiple semantics in an answer-set programming framework. In *WLP*, pages 206–210. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.5386.

[Everaere et al., 1999] Everaere, P., Konieczny, S., and Marquis, P. (1999). The strategy-proofness landscape of merging. In *In Proceedings of the Fifth European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU 99), LNAI 1638*, pages 233–244. URL: http://www.aaai.org/Papers/JAIR/Vol28/JAIR-2802.pdf.

[Faber et al., 2010] Faber, W., Pfeifer, G., and Leone, N. (2010). Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence*, In Press, Accepted Manuscript:–.

[Fitting, 1999] Fitting, M. (1999). Fixpoint semantics for logic programming - a survey. *Theoretical Computer Science*, 278:25–51. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7641.

[Gabbay et al., 2009] Gabbay, D. M., Rodrigues, O., and Pigozzi, G. (2009). Connections between belief revision, belief merging and social choice. *J. Log. Comput.*, 19(3):445–446. URL: http://people.stfx.ca/mimam/Stuff/Search%20Articles/October%203%20Search/GabbayPigozziRodrigues.pdf.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. pages 1070–1080. MIT Press. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.2912.

[Gelfond and Lifschitz, 1991] Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.7150.

[Ginsberg, 1986] Ginsberg, M. L. (1986). Counterfactuals. In *Artificial Intelligence, 30 1*. URL: http://portal.acm.org/citation.cfm?id=11572.

[Gravano et al., 2003] Gravano, L., Ipeirotis, P. G., Koudas, N., and Srivastava, D. (2003). Text joins for data cleansing and integration in an rdbms. In *In Proc. of 19th Int. Conf. on Data Engineering*, pages 729–731.

[Henrikson, 1999] Henrikson, J. (1999). Completeness and total boundedness of the hausdorff metric. *MIT Undergraduate Journal of Mathematics*, pages 69–80.

[Horrocks, 2008] Horrocks, I. (2008). Ontologies and the semantic web. *Commun. ACM*, 51(12):58–67. URL: `http://portal.acm.org/citation.cfm?id=1409377`.

[Konieczny and Pino-Pérez, 1998] Konieczny, S. and Pino-Pérez, R. (1998). On the logic of merging. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.1895`.

[Kowalski, 1974] Kowalski, R. (1974). Predicate logic as programming language. URL: `http://www.doc.ic.ac.uk/~rak/papers/IFIP%2074.pdf`.

[Lifschitz, 2008] Lifschitz, V. (2008). What is answer set programming? URL: `http://userweb.cs.utexas.edu/users/vl/papers/wiasp.pdf`.

[List and Pettit, 2002] List, C. and Pettit, P. (2002). Aggregating sets of judgments. an impossibility result. *Economics and Philosophy*, 18:89–110. URL: `http://socpol.anu.edu.au/pdf-files/W8.pdf`.

[List and Puppe, 2007] List, C. and Puppe, C. (2007). Judgment aggregation: a survey. URL: `Judgmentaggregation:asurvey`.

[Martelli and Montanari, 1982] Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282. URL: `http://portal.acm.org/citation.cfm?id=357169`.

[Naumann and Häussler, 2002] Naumann, F. and Häussler, M. (2002). Declarative data merging with conflict resolution. In *International Conference on Information Quality (IQ 2002). 2002*, pages 212–224.

[Pauly and Hees, 2006] Pauly, M. and Hees, M. V. (2006). Logical constraints on judgement aggregation. *Journal of Philosophical Logic*, 35(6):569–585. URL: `http://dx.doi.org/10.1007/s10992-005-9011-x`.

[Pettit et al., 2001] Pettit, P., SPT, R., and ANU (2001). Deliberative democracy and the discursive dilemma. Technical report, ARROW Discovery Service [http://search.arrow.edu.au/apps/ArrowUI/OAIHandler] (Australia). URL: `http://hdl.handle.net/1885/41073`.

[Pigozzi, 2006] Pigozzi, G. (2006). Belief merging and the discursive dilemma: an argument-based account to paradoxes of judgment aggregation. *Synthese*, 152(2):285–298. URL: `http://dx.doi.org/10.1007/s11229-006-9063-7`.

[Przymusinski, 1991] Przymusinski, T. C. (1991). Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.1434`.

[Rahm and Bernstein, 2001] Rahm, E. and Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *VLDB Journal*, 10:2001.

[Redl, 2010] Redl, C. (2010). Merging of biomedical decision diagrams. Master's thesis, Vienna University of Technology, Group for Knowledge-based Systems, A-1040 Vienna, Karlsplatz 13.

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41. URL: `http://portal.acm.org/citation.cfm?id=321253`.

[Shanmugasundaram et al., 2001] Shanmugasundaram, J., Kiernan, J., Shekita, E. J., Fan, C., and Funderburk, J. (2001). Querying xml views of relational data. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 261–270, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.7976`.

[Stumme and Maedche, 2001] Stumme, G. and Maedche, A. (2001). Ontology merging for federated ontologies on the semantic web. In *In Proceedings of the International Workshop for Foundations of Models for Information Integration (FMII-2001*, pages 413–418. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3255`.

[Van Emden and Kowalski, 1976] Van Emden, M. H. and Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.64.9246`.

[Veith et al., 1997] Veith, H., Eiter, T., Eiter, T., Gottlob, G., and Gottlob, G. (1997). Modular logic programming and generalized quantifiers. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-97), number 1265 in LNCS*, pages 290–309. Springer.

[Winslett, 1988] Winslett, M. (1988). Reasoning about action using a possible models approach. In *AAAI*, pages 89–93. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.2121`.