Computer Science

Michael Fink

Declarative Logic-Programming Components for Information Agents

Doctoral Thesis / Dissertation

Diplom.de

Bibliographic information published by the German National Library:

The German National Library lists this publication in the National Bibliography; detailed bibliographic data are available on the Internet at http://dnb.dnb.de .

ISBN: 9783832462529

Michael Fink

Declarative Logic-Programming Components for Information Agents

Diplom.de

Michael Fink

Declarative Logic-Programming Components for Information Agents

Dissertation / Doktorarbeit
an der Technischen Universität Wien
Fachbereich Technische Naturwissenschaften und Informatik
5 Semester Bearbeitungsdauer
September 2002 Abgabe

Diplomica GmbH
Hermannstal 119k
22119 Hamburg

Fon: 040 / 655 99 20
Fax: 040 / 655 99 222

agentur@diplom.de
www.diplom.de

ID 6252

ID 6252
Fink, Michael: Declarative Logic-Programming Components for Information Agents
Hamburg: Diplomica GmbH, 2002
Zugl.: Wien, Technische Universität, Dissertation / Doktorarbeit, 2002

Diplomica GmbH
http://www.diplom.de, Hamburg 2002
Printed in Germany

Abstract

At present, the World Wide Web faces several problems regarding the search for specific information, arising, on the one hand, from the vast number of information sources available, and, on the other hand, from their intrinsic heterogeneity. A promising approach for solving the complex problems emerging in this context is the use of information agents in a multi-agent environment, which cooperatively solve advanced information-retrieval problems. An intelligent information agent provides advanced capabilities resorting to some form of logical reasoning, based on ad hoc knowledge about the task in question and on background knowledge of the domain, suitably represented in a knowledge base.

In this thesis, our interest is in the role which some methods from the field of declarative logic programming can play in the realization of reasoning capabilities for intelligent information agents. We consider the task of updating extended logic programs (ELPs), since, in order to ensure adaptivity, an agent's knowledge base is subject to change. To this end, we develop update agents, which follow a declarative update policy and are implemented in the *IMPACT* agent environment. The proposed update agents adhere to a clear semantics and are able to deal with incomplete or inconsistent information in an appropriate way.

Furthermore, we introduce a framework for reasoning about evolving knowledge bases, which are represented as ELPs and maintained by an update policy. We describe a formal model which captures various update approaches, and define a logical language for expressing properties of evolving knowledge bases. We further investigate the semantical properties of knowledge states with respect to reasoning. In particular, we describe finitary characterizations of the knowledge evolution, and derive complexity results for our framework.

Finally, we consider a particular problem of information agents, namely *information source selection*, and develop an intelligent site-selection agent. We use ELPs for representing relevant knowledge and for declarative query analysis and query abstraction. We define syntax and semantics of declarative site-selection programs, making use of advanced methods from answer set programming for priority handling and quantitative reasoning. A site selection component is implemented on top of the `DLV` KR system and its `plp` front-end for prioritized ELPs. We report experimental results for this implementation, obtained using a representative example from a movie domain.

Kurzfassung

Die Suche nach spezifischer Information im Internet steht zur Zeit einer Reihe von Problemen gegenüber, die sich einerseits auf die große Anzahl verfügbarer Informationsquellen und andererseits auf deren immanente Heterogenität zurückführen lassen. Ein vielversprechender Ansatz zur Lösung der komplexen Probleme, die in diesem Zusammenhang auftreten, ist der Einsatz von Informationsagenten in Multi-Agenten Systemen, in denen mehrere Informationsagenten kooperieren, um gemeinsam schwierige Aufgaben der Informationsbeschaffung zu lösen. Ein intelligenter Informationsagent entwickelt dabei besondere Fähigkeiten, indem er logisches Schließen auf eine Wissensbasis anwendet, die auf formalem Wissen über die jeweilige Aufgabe und auf Hintergrundwissen über den Problembereich basiert.

In dieser Dissertation wird untersucht, welche Rolle Methoden der deklarativen logischen Programmierung in der Entwicklung von Komponenten für intelligente Informationsagenten spielen können. Es wird zunächst das Problem betrachtet, eine Wissensbasis, die durch sogenannte *Erweiterte Logische Programme* (ELPs) repräsentiert ist, entsprechend zu aktualisieren. Dies ist deshalb von besonderer Bedeutung, da von einem intelligenten Informationsagenten erwartet wird, daß er sich an Änderungen seines Umfeldes entsprechend anpaßt. Es werden sogenannte „Update Agents“ entwickelt und als *IMPACT*-Agenten implementiert, die diese Aufgabe lösen, indem sie einer deklarativen Strategie folgen. Die vorgeschlagenen Agenten zeichnen sich durch ihre klar definierte Semantik aus und sind darüberhinaus in der Lage, mit unvollständiger sowie inkonsistenter Information umzugehen.

Desweiteren wird ein theoretisches System eingeführt, welches das logische Schließen über dynamische Wissensbasen ermöglicht, welche als ELPs repräsentiert sind und anhand einer deklarativen Strategie aktualisiert werden. In diesem formalen Modell lassen sich verschiedenste Update-Ansätze und Methoden ausdrücken. Eine eigens definierte logische Sprache ermöglicht es, Eigenschaften derartiger Wissensbasen formal auszudrücken und durch formales Schließen zu verifizieren. Da letzteres aber rechnergestützt nur möglich ist, wenn die „Evolution“ der Wissensbasis durch eine endliche Anzahl verschiedener Zustände beschreibbar ist, werden endliche Charakterisierungen herausgearbeitet und zur Untersuchung der computationalen Komplexität des Systems herangezogen.

Zuletzt widmen wir uns einer konkreten Aufgabe von Informationsagenten, nämlich der Auswahl geeigneter Informationsquellen, und entwickeln dafür einen intelligenten „Site-Selection Agent“. Dabei werden ELPs nicht nur zur Repräsentation relevanten Wissens verwendet, sondern auch um Anfragen deklarativ zu analysieren und eine abstrakte Repräsentation einer Anfrage zu berechnen. Syntax und Semantik deklarativer „Site-Selection Programme“ werden definiert, indem

auf fortgeschrittene Methoden der Answer-Set Programmierung zurückgegriffen wird, die der Behandlung von Prioritäten und dem quantitativen Schließen dienen. Unter Zuhilfenahme des Wissensverarbeitungs-Systems `DLV` und dessen Front-end `plp` für ELPs mit Prioritäten wird eine „Site-Selection“ Komponente für intelligente Informationsagenten implementiert. Experimentelle Ergebnisse werden anhand einer repräsentativen Beispielanwendung aus dem Kinofilmbereich gewonnen und analysiert.

Acknowledgements

I am grateful to a number of people who contributed to this thesis in various ways. First of all, I owe a great debt to my supervisors Thomas Eiter and Hans Tompits. Even from abroad they always had an ear for (and, of course, answers to) my questions ranging from technical problems to presentational matters. I am much obliged to Thomas Eiter for giving me the opportunity to work within the Knowledge-based Systems Group at the Vienna University of Technology and a special thanks to Hans Tompits for making me aware of this position.

I would also like to thank Piero Bonatti, Marco Cadoli, Jürgen Dix, Thomas Lukasiewicz, and V.S. Subrahmanian (in alphabetical order) for deepening my interests in logic programming and nonmonotonic reasoning in outstanding lectures and fruitful scientific discussions.

I want to acknowledge T.J. Rogers for disclosing the secrets of the *IMPACT* agent development environment for me. I also benefitted a lot from work by Torsten Schaub and his student Sven Thiele, who extended the `plp` front-end for our needs, and Wolfgang Faber was a great aid in all questions concerning `DLV`. Many thanks at this point to Roman Schindlauer for his collaboration regarding implementation tasks.

Giuliana Sabbatini completed our project team and I really enjoyed the time being room-fellows. Special thanks to her and all members of our department for the warm and amicable working atmosphere. Moreover, I want to express my gratitude to our secretary, Elfriede Nedoma, and our technician, Matthias Schlögel, who have been a constant source of assistance.

Last, but never least, I want to thank my family, my close friends, and my beloved Bettina. They encouraged me, tolerated my moods, and cared for distraction in bad times and during the “hot stage” in the last weeks. With their love they make my life a pleasure.

This thesis was supported by the Austrian Science Fund (FWF) under grants P13871-INF and P14781-INF.

Contents

List of Figures

List of Tables

1 Introduction

In the past decade, with the World Wide Web entering our daily life, a wealth of information has become available to a large group of users. The Internet hype was carried by a wave of optimism and expectations in the radically new forms of communication, information dissemination, and information retrieval the new technology provided. To date, it seems that while the Web is still highly appreciated for communicating and publishing, when searching the Web for information, users are often dissatisfied with the means for searching and the results obtained.

A user faces several problems when looking for desired information on the Web, which arise from the vast amount of available information sources and from their heterogeneity, since standards are missing. First of all, the user has to identify the relevant sources which should be queried, since bounds on resources, time, and/or cost of services usually do not permit to query *all* sources which are available. The generation of a query plan is often quite expensive and cannot be optimal when the user has no additional information about the knowledge contained in each source, but merely a short description of it. Then, once the user has made up his or her plan, for each source the query must be formulated in the appropriate way, depending on the interfaces available, as well as on the data organization and presentation. Furthermore, the user must possibly adapt the query more than once for each source in order to get the desired information, and must learn several different query approaches. Furthermore, it may happen that some sources provide as a result only partial or incomplete information to a query. The user has then to merge all data retrieved, taking into account that the information provided by different sources may be inconsistent. In such a situation, the user needs some notion of reliability for the sources in order to choose the proper information.

There is need for a suitable information processing infrastructure relieving the user from these tedious tasks. A promising approach for solving the complex problem is the use of a *multi-agent system* for accessing several heterogeneous information sources. The user is presented a uniform interface for accessing all available services and information sources, without having to bother with the heterogeneity underneath. It is the system as a whole which takes care of searching the appropriate sources, accessing them, and returning to the user the required information, and this to an extent as complete and consistent as possible.

The realization of such systems requests for special functionalities and capabilities, which emerge from specialized tasks like search and assessment of information sources, query planning, and information merging and fusion, thereby dealing with incomplete information and inconsistencies. Such capabilities are provided as services on request by various kinds of *information agents*,

which usually form a society of agents for cooperatively solving complex information-retrieval problems. It is not hard to imagine that an advanced approach to any of these problems must involve, in some form, logical reasoning tasks, based on ad hoc knowledge about the task in question and on background knowledge of the domain, suitably represented in a knowledge base. It is the goal of this thesis to investigate how methods from declarative logic programming can be employed to allow information agents to solve some of their tasks more "intelligently".

1.1 Intelligent Information Agents

In this thesis, we focus on *information agents* (sometimes also called *middle agents* [46]), a special kind of so-called software agents (see [122], for an extensive overview of software agent programming approaches). The term information agent is broadly used in the literature and several types of information agents, as well as facilities for their cooperation, have been suggested and implemented. Following [76], they can be classified as follows:

Facilitators: Agents which take control over a set of subordinated agents and coordinate the services they offer and the use of the resources they require.

Brokers: Agents often used for matching between a set of different data sources and user requests. Brokers receive requests, look for relevant sources matching them, and then perform actions using services from other agents (combining them with their own resources or information).

Mediators: In the mediator approach [131], meta-knowledge about a number of other agents (sometimes called *provider agents*) is available to the mediator, which exploits it to create higher-level services (not provided by the underlying agents) for user applications. These new services result by the combination and merging of low-level services on the basis of the comprehensive meta-information which the mediator has. In a sense, mediators may be seen as enhanced, high-level brokers.

Yellow Pages: A yellow pages dictionary helps users and other agents in finding the right providers (other agents or information sources) for the kind of service they need, possibly performing a match between advertised services, ontology of the domain in question and service requests.

Blackboards: A blackboard serves as a temporal repository for service requests which remain to be processed. Agents offering some service can access the blackboard and look for (and retrieve from it) service requests which they can satisfy.

We are particularly interested in information agents acting as mediators. Thus, we assume that the agent has knowledge about the application domain of interest, and some meta-knowledge about the contents and features of the distributed, heterogeneous information sources the system has access to.

In order to satisfy a user request, an information agent may have to solve various subgoals, such as identifying and possibly ranking relevant information sources; retrieving the required information (or ask some appropriate provider agent for it); processing the information returned by the sources by combining, merging, and integrating them; optimizing the number of accessed sources, or the total cost and time required for the search, or the precision and completeness of the results.

For fulfilling these tasks, an agent has to make decisions about what to do or how to proceed at certain points. An *intelligent information agent* takes its decisions by reasoning using the knowledge that it possesses about the application and the particular request to be answered. Rather than having decision mechanisms implicitly hard-coded into the agent's program, we assume that the agent's decision making facilities are modularized and attached to it as *reasoning components*. The latter can be realized in many ways, by using one of the numerous approaches that have been developed in the agent and AI literature.

1.1.1 Problems and challenges

Intelligent information agents are supposed to provide advanced capabilities which help in improving the quality of query results. It is these capabilites that might be realized as, or supported by, reasoning components. We list some of these capabilities below, without claiming that the list is exhaustive:

- **Decompose a request** in its "atomic" parts on the basis of available knowledge about the information sources, and reformulate it according to the corresponding data structure. The decomposition task can be of varying complexity and can generate a set of possible decompositions, based on the information about the underlying sources and the domain of interest [102, 103, 104, 37, 91, 23, 24, 26, 17, 18, 82, 83, 120, 6, 7, 8, 9].
- **Integrate the query** with additional user information, if available. This could require asking first a *profiling agent* for the user profile, habits, rights and interests.
- **Select the information sources** to be queried, using the meta-knowledge about them and about the application domain to determine which of the sources contain relevant data to answer the query [102, 103, 104, 37, 91, 23, 24, 26, 8, 9, 17, 18, 82, 83, 120]. If possible, determine further preferred information sources to be queried.
- **Create a query plan.** Determine in which order sub-queries are to be performed [102, 103, 104, 17, 18], on the basis of query decomposition, of the data available in every source, and of preference over sources, in order to optimize the execution.
- **Execute the plan**, asking the corresponding provider agents for the required services and waiting for their answers, possibly adapting the query plan dynamically to run-time information and to the answers obtained so far [102, 103, 17, 18, 98].
- **Compose and merge the answers.** This task can have varying complexity, from simply collecting all the answers and passing them to the interface agent without processing them,

to organizing the retrieved data in some form (e.g., eliminating multiple instances of the same information), to merging them in a single comprehensive answer, and so on [37, 91, 23, 24, 8, 9, 82, 83, 120].

- **Detect and possibly remove inconsistencies** among retrieved data [37, 91, 8, 9] on the basis of an inconsistency removal strategy and of meta-knowledge about the sources, or meta-knowledge contained in the answers themselves.

- **Integrate incomplete information** by means of internal reasoning capabilities [82, 83, 120].

- **Start a learning procedure** in order to improve or update internal models and reasoning rules [17, 18], perhaps providing input to the profiling agent as well.

1.1.2 Systems and frameworks

Some of the tasks for intelligent information agents identified above have already been widely addressed and some feasible solutions have been developed. Following the grouping suggested in [105], existing intelligent Web systems have their main focus in some of the following fields:

User modeling and profiling: addressing the issue of deploying adaptive user interfaces and recommendation systems, integrating user queries on the basis of the user profile or directly suggesting the user items of interest.

Analysis and preprocessing of information sources: building up, on the basis of an application domain description, the meta-knowledge on which reasoning and decision making is based.

Information integration and information management: covering a wide area of applications and different problems.

Especially in the field of information integration a large number of systems and frameworks has been developed, e.g., the Information Manifold [102, 103, 104], Carnot [91, 37], InfoSleuth [23, 24, 26], *HERMES* [8, 9, 10, 127], *IMPACT* [128, 19, 69, 70], *SIMS* [17, 18], or *TSIMMIS* [82, 83, 120] (for an overview of these systems cf. [64]). They are highly relevant, since the kind of problems tackled by them and suggestions for solutions are of interest in the whole field of heterogeneous information integration. However, many of these systems use ad hoc procedural techniques, rather than declarative methods – especially declarative logic programming methods – in which we are interested in. For a comprehensive overview of existing systems in the field of information integration and of the role of computational logic in some of them, we refer to [52]; for hints and relevant literature on user modeling and preprocessing of information sources, see [105].

1.2 Declarative Methods

By *declarative methods*, as opposed to procedural methods, we mean methods that allow us to specify a problem in a formal language and to use an inference mechanism in order to solve it, instead of operationally constructing, i.e., "programming", the solution of the problem. In particular, by *declarative logic programming* we mean logic programming techniques, where the ordering of rules, as well as the ordering of atoms or literals in their bodies, have no influence on the inference procedure and thus on the models obtained (e.g., *answer set programming* [118] as opposed to Prolog).

Our motivation to employ declarative methods – and in particular answer set programming techniques – for developing reasoning components for information agents is driven by the following advantages:

- They provide a clear formal semantics to the reasoning component.
- Changes in the specification are easily incorporated by modifying or adding suitable rules or constraints, without the need for re-designing the entire component, as might be the case, e.g., in procedural languages.
- Answer set programming is capable of handling incomplete information and performing nonmonotonic inferences, which, arguably, is an inherent feature of the problem domain.
- Finally, the declarative nature of the answer set semantics formalism permits the coupling with other logic-based components, e.g., sophisticated ontology tools and their reasoning engines, which may be employed in order to providing advanced features.

In search for feasible or improved solutions to the challenges and problems of intelligent information agents, given in Section 1.1.1, some central reasoning sub-tasks can be identified, which the agent itself must have or be able to access in form of reasoning components. For some of these reasoning tasks, listed below, declarative methods seem to be promising.

- **Priority handling.** Dealing with priority in the logic programming context has received considerable attention, see, e.g., [86, 30, 93, 77, 47, 29, 32, 31, 14]. Priority information needs to be encoded in the knowledge base of the agent, possibly explicitly in the object language in which the knowledge itself is expressed. The preference relation may be static or dynamic, in the latter case reasoning procedures have to account for possible changes.
- **Revision and update.** The knowledge base of the agent can be subject to change on the basis of information from the environment (the application domain itself, other agents inside or outside the system, etc.), in order to incorporate changes in the outside world or changes in the knowledge about it [29, 111, 101, 92, 123, 94, 78, 79, 11, 12, 13, 14, 15]. The epistemic state and the intended belief set of the agent have thus to be continuously revised, depending on some specified update policy, which could be in turn explicitly described in the knowledge base and maybe dynamically updated too.

- **Inconsistency removal.** The agent should in the presence of conflicts in the knowledge base be able to detect them and to find an acceptable *fall-back* knowledge configuration, in order to ensure that the decision making process is not stopped or inconsistent as a whole. The fall-back configuration is usually required to preserve "as much as possible" of the available knowledge.

- **Decision making with incomplete information.** Under the manifestation of incomplete information, some reasoning and deduction capabilities may provide candidate hypotheses for the missing information pieces, in order to ensure that the process of decision making can derive plausible results.

- **Temporal reasoning.** The evolution of a dynamic knowledge base could be subject to specifications [15], and corresponding forms of reasoning about this evolution could be provided for ensuring that the agent's behavior is appropriate, e.g., that some undesired status cannot be reached [71, 43].

- **Learning.** Based on the history of the agent (sequence of changes in its knowledge base, or sequence of observations), some form of inductive learning could be implemented [123, 94, 50].

A number of different models and methods for knowledge representation and reasoning have been developed in the past, which may be used for this purpose, e.g., description logics, abduction, induction, and argumentation (cf. [122] for a broad overview and references). However, the focus of this thesis is on the role which some methods from the field of declarative logic programming can play in the realization of reasoning components for information agents. In particular, we are interested to see how they can be used, extended, and further developed for the needs of this domain of application. For an overview of methods from declarative logic programming aimed at the above reasoning tasks the reader is referred to [64]

1.3 Outline

The main problems addressed in this thesis are the following. First, we consider the task of updating logic programs. Since an information agent is situated in an environment which is subject to change, it is required to adapt over time, and to adjust its decision making. For agents utilizing logic programming techniques for representing (parts of) their knowledge, this requires the agent to be able to update logic programs accordingly, in order to ensure adaptivity. Several approaches for updating nonmonotonic logic programs, have been proposed, cf. [11, 12, 55, 56, 121, 79, 94, 101]. Towards a reasoning component for updating the knowledge base of an information agent using extended logic programs for knowledge representation, we choose one of the approaches, viz. *update answer set semantics* [55, 56, 121], develop algorithms for its implementation, and realize a tool for application.

Besides an underlying update semantics, which specifies how new, possibly inconsistent information is to be incorporated into the knowledge base, an agent needs to have a certain *update*

policy, i.e., a specification of how to react upon the arrival of an update. The issue of how to specify change requests for knowledge bases has received growing attention more recently and suitable specification languages for nonmonotonic logic programs have been developed [15, 99, 100, 16, 57, 121, 62, 60]. We choose an approach, the language EPI [57, 121, 62, 60], which is not only independent of the underlying update semantics, but can also be "agentized" in the *IMPACT* [128, 19, 69, 70] agent architecture in a straightforward way. By this means, we build so-called *update agents* by "plugging-in" the implementation of update answer set semantics, developed before.

Second, we address the task of temporal reasoning. Every intelligent agent is supposed to be able to reason about future states, e.g., consequences of its behavior. While the update component outlined above, enables an information agent, built on declarative logic programming techniques, to query its knowledge state after a number of updates have occurred, it does not enable the agent to answer queries such as whether a particular fact will be true in all possible future knowledge states. Analogous issues, called *maintenance* and *avoidance*, have been recently studied in the agent community [133]. However, reasoning about an evolving knowledge base, maintained using an update policy, has not yet been formally addressed. Thus, we aim at a framework for expressing reasoning problems over such evolving knowledge bases, where we generalize from the update policy – and of course of the update semantics – used. Within the framework, it shall be possible to capture different approaches of incorporating updates into logic programs, still paying attention to the specific nature of the problem. Furthermore, it should be possible to evaluate a formula, which specifies a desired evolution behavior, across different realizations of update policies based on different grounds.

Third, we consider a particular problem of information agents, namely information source selection, and develop an intelligent *site selection agent* building on declarative methods. Given a query by the user and a collection of information sources, the agent's task is to select a source for answering the query such that the utility of the answer, in terms of quality of the result and other criteria, like, e.g., costs, is as large as possible for the user. An intelligent solution to this problem needs to combine several aspects, such as basic properties of the information sources, knowledge about their contents, and knowledge about the particular application domain. To this end, our site selection component will incorporate advanced methods from declarative logic programming for priority handling [30, 49, 95] and quantitative reasoning [34].

Main results. The main contributions of this thesis can be summarized as follows:

(1) We develop algorithms that allow for update answer set semantics to be implemented by the use of existing logic programming systems as an underlying reasoning engine. Based on these algorithms, we develop a tool, called upd, which is conceived as a front-end to the state-of-the-art solver DLV [68, 54]. The implementation allows for different modes of reasoning and handles also refinements of the semantics involving certain minimality-of-change criteria.

(2) We develop update agents, which deploy the update front-end as their basic reasoning component and follow a declarative update policy. To this end, we implement the EPI framework for update specification by developing appropriate software packages for the *IMPACT* agent

platform. The implementation is generic in the sense that it allows for other implementations of update semantics for logic programs to be plugged in. Furthermore, agents can make use of additional features of *IMPACT*, e.g., enriching the policy language with deontic modalities.

(3) We introduce a generic formal model in which various approaches for updating extended logic programs can be expressed. In particular, we introduce the concept of an *evolution frame*, whose components serve to describe the evolution of knowledge states. Based on evolution frames, we define the syntax and the semantics of a logical language for reasoning about evolving knowledge bases, such that properties of an evolving knowledge base can be formally stated and evaluated in a systematic fashion, rather than ad hoc.

(4) We investigate semantical properties of knowledge states for reasoning. In particular, since in principle a knowledge base may evolve forever, we are concerned with finitary characterizations of evolution. To this end, we introduce various notions of equivalence between knowledge states, and show several filtration results. As an application, we establish this for evolution frames which model policies in the EPI framework for logic program updates using the answer set semantics, as well as for LUPS and LUPS* policies [15, 16, 99] under the dynamic stable model semantics [12].

(5) We derive complexity results for reasoning. Namely, we consider the problem of deciding whether a given property, expressed by a formula in our logical language, holds for a given knowledge state in a given evolution frame. While the problem is undecidable in general, we single out several cases in which the problem is decidable, adopting some general assumptions about the underlying evolution frame. In this way, we identify meaningful conditions under which the problem ranges from PSPACE up to 2-EXPSPACE complexity. We again apply this to the EPI framework, showing that its propositional fragment has PSPACE complexity.

(6) We develop a reasoning component for intelligent information source selection, using extended logic programs (ELPs) [85] to represent rich descriptions of the information sources, an underlying domain theory, and queries in a formal language. We perform query analysis by ELPs and compute *query abstractions*. At the heart, a declarative *site selection program* represents *both qualitative and quantitative* criteria (e.g., site preference and costs).

(7) We consider the interesting and, to our knowledge, novel issue of *contexts* in logic programs. Structured data items require a careful definition of the selection semantics whilst inheritance-based approaches, such as [33], do not apply here. Furthermore, implicit priorities, derived from context information, must be combined with explicit user preferences from the selection policy, and arising conflicts must be resolved.

(8) We have implemented a site selection agent and performed experiments in an example application from the movie domain. Our example application comprises several XML databases, wrapped from movie databases available on the Web, and handles queries in XML-QL. Experiments that we have conducted show that the system behaved intuitively on a number of natural queries, some of which require reasoning from the background knowledge to identify the proper selection.

The remainder of this thesis is organized as follows. In the next chapter we briefly introduce extended logic programs, the basic formalism used for knowledge representation throughout this thesis, and the *IMPACT* agent architecture, which represents the platform underlying our agent implementations. Chapter 3 is devoted to update agents. There, we first introduce update answer set semantics for updating extended logic programs, and then we address its implementation. In the second part of the chapter, the EPI framework for update specifications is outlined leading to the realization of update agents. In Chapter 4, we deal with reasoning about the evolution of nonmonotonic knowledge bases. The formal definitions of an evolution frame and a formal language for expressing properties of an evolving knowledge base serve as a starting point for investigations on equivalence relations over knowledge states, which are useful in order to obtain finite characterizations of the transition graph, and on the computational complexity of reasoning. Site selection agents are the topic of Chapter 5. After giving an overview of the site selection process and architecture, we formally define our approach to knowledge-based site selection, before we turn to its implementation and application serving as a basis for the experimental results reported. Finally, Chapter 6 concludes the thesis with a summary and some general remarks.

All original results contained in this thesis have been published as refereed papers in journals and proceedings of international conferences. The update semantics and its implementation are reported in *Theory and Practice of Logic Programming* [63] and appeared in preliminary form in the proceedings of JELIA 2000 [55]. The EPI language and its agentization in *IMPACT* have been presented at IJCAI 2001 [57] as well as at the AISB 2001 Symposium on Adaptive Agents and Multi-Agent Systems [58]. Most results of Chapter 4 are contained, in preliminary form, in the proceedings of LPAR 2001 [59], and the site-selection approach of Chapter 5 has been presented at KR 2002 [61].

2 Preliminaries

2.1 Declarative Logic Programming

The key method for knowledge representation used throughout this thesis is declarative logic programming. In particular, we deal with extended logic programs [85], i.e., sets of rules, built over a set $\mathcal{A}$ of (first-order) atoms where both default negation not (often also referred to as weak negation and also called negation as failure) and strong negation $\neg$ (sometimes also called classical negation) is available. Facts are represented by *literals*. A literal, L, is either an atom A (a *positive literal*) or a strongly negated atom $\neg A$ (a *negative literal*). For a literal L, the *complementary literal*, $\neg L$, is $\neg A$ if $L = A$, and A if $L = \neg A$, for some atom A. For a set S of literals, we define $\neg S = \{\neg L \mid L \in S\}$. We also denote by $Lit_{\mathcal{A}}$ the set $\mathcal{A} \cup \neg\mathcal{A}$ of all literals over $\mathcal{A}$. A literal preceded by the default negation sign not is said to be a *weakly negated literal*. In contrast to strong negation $\neg L$, expressing the fact that L is false, the intuition of weak negation is, that $not\ L$ is true *if we cannot assert that L is true*, i.e., if either L is false or we do not know whether L is true or false. We first consider the propositional case, i.e., $\mathcal{A}$ is a set of propositional atoms, while first-order programs will be introduced in Section 2.1.3.

2.1.1 Syntax

A *rule*, r, has the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n,$$

where L_i, $0 \leq i \leq n$, are literals. We call L_0 the *head* of r and $B(r) = \{L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n\}$ the *body* of r. Furthermore, we will often use $H(r)$ to denote the head of rule r. If $B(r) \neq \emptyset$, we also allow the case where L_0 may be absent. We define $B^+(r) = \{L_1, \ldots, L_m\}$ and $B^-(r) = \{L_{m+1}, \ldots, L_n\}$. The elements of $B^+(r)$ are referred to as the *prerequisites* of r. Intuitively, rule r means that we can conclude L_0 if (i) $L_1, \ldots, L_m$ are known and (ii) $L_{m+1}, \ldots, L_n$ are *not* known. We employ the usual conventions for writing rules like $L_0 \leftarrow B_1 \cup B_2$ or $L_0 \leftarrow B_1 \cup \{L\}$ as $L_0 \leftarrow B_1, B_2$ and $L_0 \leftarrow B_1, L$, respectively.

If r has an empty head, then r is a *constraint*; if $H(r) = \{L_0\}$ and the body of r is empty, then r is a *fact*. In the latter case, slightly abusing notation, r is often simply represented by its head literal L_0. If $n = m$ (i.e., if r contains no default negation), then r is a *basic rule*.

An *extended logic program* (ELP), P, is a (possibly infinite) set of rules. If all rules in P are basic, then P is a *basic program*. We say that P is an extended logic program *over* $\mathcal{A}$ if all atoms occurring in the rules of P are in a certain specified set $\mathcal{A}$ of atoms. Usually, $\mathcal{A}$ will simply be understood as the set of *all* atoms occurring in P. We denote by $\mathcal{L}_{\mathcal{A}}$ the set of all rules constructible using the literals in $Lit_{\mathcal{A}}$.

Example 1 Consider a knowledge base KB represented as an ELP consisting of the following rules:

$$\begin{aligned} KB \;=\; \{ & r_1 : night, \leftarrow , \\ & r_2 : \; tv\_on \leftarrow , \\ & r_3 : \; watch\_tv \leftarrow tv\_on, \\ & r_4 : \; sleep \leftarrow night, not\ tv\_on \; \}. \end{aligned}$$

KB consists of two facts, r_1 and r_2, capturing a situation where it is night and my TV is on. The basic rule r_3 states that whenever my TV is on, then I am watching TV. Finally, rule r_4 specifies to infer, by default, that I am asleep, whenever it is night and it cannot be concluded that my TV is on. □

A set of literals is *consistent* iff it does not contain a complementary pair A, $\neg A$ of literals. Consistent sets of literals are also referred to as *interpretations*.

A literal L is *true* in an interpretation I (symbolically $I \models L$) iff $L \in I$, and *false* otherwise. Given a rule r, the body $B(r)$ of r is true in I iff (i) each $L \in B^+(r)$ is true in I and (ii) each $L \in B^-(p)$ is false in I. In other words, $B(r)$ is true in I iff $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. We write $I \models B(r)$ to express that $B(r)$ is true in I. Rule r is true in I iff $H(r)$ is true in I whenever $B(r)$ is true in I. In particular, if r is a constraint, then r is true in I if $B(r)$ is not true in I. The fact that r is true in I will be denoted by $I \models r$. Likewise, for a program P, $I \models P$ means that $I \models r$ for all $r \in P$. In this case, I is said to be a *model* of P.

2.1.2 Semantics

Since rules may include weak negation, they are more expressive than ordinary Horn clauses. This is the intuitive reason why we cannot always assign a unique set of consequences to an ELP. Another effect is that there exist several semantics of ELPs [53]. One of the most important is the concept of an answer set [85], introduced below, which generalizes the concept of a stable model for *general logic programs* [84] (i.e., programs not containing classical negation, $\neg$).

Let r be a rule. Then r^+ denotes the basic rule obtained from r by deleting all weakly negated literals in the body of r, i.e., $r^+ = H(r) \leftarrow B^+(r)$. Furthermore, we say that rule r is *defeated* by a set of literals S if some literal in $B^-(r)$ is true in S, i.e., if $B^-(r) \cap S \neq \emptyset$. As well, each literal in $B^-(r) \cap S$ is said to *defeat* r.

The *reduct*, P^S, of a program P *relative to* a set S of literals is defined by

$$P^S = \{r^+ \mid r \in \Pi \text{ and } r \text{ is not defeated by } S\}.$$

In other words, P^S is obtained from P by (i) deleting any $r \in P$ which is defeated by S and (ii) deleting each weakly negated literal occurring in the bodies of the remaining rules. An interpretation I is an *answer set* of a program P iff it is a minimal model of P^I, i.e., $I \models P^I$ and no model I' of P^I exists, such that I is a proper subset of I'. Observe that any answer set of P is *a fortiori* a model of P. The set of all *generating rules* of an answer set S from P is given by $GR(P, S) = \{r \in P \mid S \models B(r)\}$.

By $\mathcal{AS}(P)$ we denote the collection of all answer sets of P. If $\mathcal{AS}(P) \neq \emptyset$, then P is said to be *satisfiable*, otherwise P is *inconsistent*. The answer set semantics is due to Gelfond and Lifschitz [85] and, thus, the reduct P^I is often called the *Gelfond-Lifschitz reduct*.

Example 2 Reconsider the knowledge base KB of Example 1. Given the interpretation $S = \{night, tv\_on, watch\_tv\}$, the reduct KB^S consists of the rules r_1, r_2, and r_3. It is easily verified that S is a minimal model of KB^S. Hence, S is an answer set of KB. □

2.1.3 First-order programs

It is quite straightforward to extend the notion of answer sets to the case where variables may occur in rules. This is done in the usual way by defining the semantics of programs with variables in terms of the semantics of ground programs.

Let $\mathcal{A}$ be some countable first-order alphabet without equality and let $\mathcal{P}$ be a set of predicate symbols from $\mathcal{A}$. By a *first-order program*, P, *over* $\mathcal{P}$ we understand a set of rules with $\mathcal{P}$ being the totality of predicate symbols occurring in the rules of P.

For a list $\mathbf{X} = X_1, \ldots, X_n$ of variables, we write $r(\mathbf{X})$ to indicate that rule r may contain the variables $X_1, \ldots, X_n$. Accordingly, instances of $r(\mathbf{X})$ will be denoted by $r(\mathbf{t})$, where $\mathbf{t} = t_1, \ldots, t_n$ is a list of terms and $r(\mathbf{t})$ results from $r(\mathbf{X})$ by uniformly replacing occurrences of X_i by t_i ($1 \leq i \leq n$). We retain our convention of denoting the head of rule r by $H(r)$ and the body of r by $B(r)$.

The *Herbrand universe* of a program P consists of all terms constructible from the constants and function symbols occurring in P. An instance $r(\mathbf{t})$ of a rule $r(\mathbf{X}) \in P$ is *ground* iff $\mathbf{t}$ is a list of terms from the Herbrand universe of P. The *ground instantiation*, P^*, of P consists of all ground instances of the rules from P. As already mentioned above, the ground instantiation P^* of the first-order program P determines the answer sets of P. More specifically, the set $\mathcal{AS}(P)$ of all answer sets of P is identified with the set $\mathcal{AS}(P^*)$, where the answer sets of ground programs are defined as for the propositional case.

2.1.4 Belief set and epistemic state

If a logic program P is regarded as the *epistemic state* of an agent, then the given semantics can be used for assigning a *belief state* to any epistemic state P in the following way.

Let $I \subseteq Lit_{\mathcal{A}}$ be an interpretation. Define

$$Bel_{\mathcal{A}}(I) = \{r \in \mathcal{L}_{\mathcal{A}} \mid I \models r\}.$$

Furthermore, for a set $\mathcal{I}$ of interpretations, let $Bel_{\mathcal{A}}(\mathcal{I}) = \bigcap_{i \in \mathcal{I}} Bel_{\mathcal{A}}(I)$.

Definition 1 *For a logic program P, the belief state, $Bel_{\mathcal{A}}(P)$, of P is given by $Bel_{\mathcal{A}}(P) = Bel_{\mathcal{A}}(\mathcal{AS}(P))$, where $\mathcal{AS}(P)$ is the collection of all answer sets of P.*

We write $P \models_{\mathcal{A}} r$ if $r \in Bel_{\mathcal{A}}(P)$. As well, for any program Q, we write $P \models_{\mathcal{A}} Q$ if $P \models_{\mathcal{A}} q$ for all $q \in Q$. Two programs, P_1 and P_2, are *equivalent* (modulo the set $\mathcal{A}$), symbolically $P_1 \equiv_{\mathcal{A}} P_2$, iff $Bel_{\mathcal{A}}(P_1) = Bel_{\mathcal{A}}(P_2)$. Usually we will drop the subscript "$\mathcal{A}$" in $Bel_{\mathcal{A}}(\cdot)$, $\models_{\mathcal{A}}$, and $\equiv_{\mathcal{A}}$ if no ambiguity can arise.

2.1.5 Computational complexity

We appeal to the basic concepts of complexity theory as they can be found in [115] ([97] is also a good source for basic complexity classes; for a background on complexity results in logic programming, cf. [124, 66, 45]; for complexity results of nonmonotonic logics in general see [89, 36]). Let us briefly recall the definitions of relevant complexity classes.

We consider complexity classes of the form TIME(f) (deterministic time), NTIME(f) (nondeterministic time), SPACE(f) (deterministic space), and NSPACE(f) (nondeterministic space), where $f : \mathbb{N} \to \mathbb{N}$, is a nondecreasing, polynomial-time computable function. The class TIME$(f(n))$ contains all decision problems which are solvable by a deterministic Turing machine in at most $f(n)$ steps, while NTIME$(f(n))$ contains all decision problems which are solvable by a nondeterministic Turing machine in time bounded by $f(n)$. Similarly, SPACE$(f(n))$ contains all decision problems solvable by a deterministic Turing machine requiring at most space $f(n)$ and NSPACE$(f(n))$ contains all decision problems solvable by a nondeterministic Turing machine using space bounded by $f(n)$.

We define the class P, consisting of all decision problems which are solvable in polynomial time using a deterministic Turing machine, as $\mathrm{P} = \bigcup_{k>0} \mathrm{TIME}(n^k)$. Correspondingly, the class $\mathrm{NP} = \bigcup_{k>0} \mathrm{NTIME}(n^k)$ consists of all decision problems which are solvable in polynomial time using a nondeterministic Turing machine. Moreover, Σ_2^P is the class of all decision problems solvable by a nondeterministic Turing machine in polynomial time with access to an oracle for problems in NP (Σ_2^P is also written as $\mathrm{NP}^{\mathrm{NP}}$). Furthermore, coNP refers to the class of problems whose complementary problems are in NP, and Π_2^P contains the complements of the problems in Σ_2^P.[1] All the mentioned classes belong to the *polynomial hierarchy*: NP and coNP are at the first level of the polynomial hierarchy, and Σ_2^P and Π_2^P are the second level. As well, $\mathrm{NP} \subseteq \Sigma_2^P$ and $\mathrm{coNP} \subseteq \Pi_2^P$. It is widely held that these inclusions are proper.

Moreover, PSPACE is the class of all decision problems which are solvable in polynomial space using a deterministic Turing machine (i.e., $\mathrm{PSPACE} = \bigcup_{k>0} \mathrm{SPACE}(n^k)$). Similarly, NPSPACE is the class of all decision problems which are solvable in polynomial space using a nondeterministic Turing machine (i.e., $\mathrm{NPSPACE} = \bigcup_{k>0} \mathrm{NSPACE}(n^k)$). It is well known that NPSPACE =

[1] Two decision problems, D_1 and D_2, are complementary (or, D_1 and D_2 are complements of each other) if it holds that I is a yes-instance of D_1 exactly if I is a no-instance of D_2.

PSPACE, as well as it holds that every decision problem in the polynomial hierarchy is solvable in PSPACE. Furthermore,

$$\text{EXPSPACE} = \bigcup_{k>0} \text{SPACE}(2^{n^k}) \text{ and 2-EXPSPACE} = \bigcup_{k>0} \text{SPACE}(2^{2^{n^k}}),$$

are the classes of decision problems solvable in (single) exponential space, respectively double exponential space, using a deterministic Turing machine. Similarly,

$$\text{EXPTIME} = \bigcup_{k>0} \text{TIME}(2^{n^k}) \text{ and 2-EXPTIME} = \bigcup_{k>0} \text{TIME}(2^{2^{n^k}}),$$

are the exponential time, respectively double exponential time, classes.

Finite, propositional extended logic programming resides at the first level of the polynomial hierarchy [110, 45], i.e., determining whether an extended logic program P has an answer set is NP-complete, and determining whether $L \in Bel(P)$ for some literal L is coNP-complete.

2.2 *IMPACT* Agents

As an agent framework for implementations, the *Interactive Maryland Platform for Agents Collaborating Together* (*IMPACT*) ([128, 19, 69, 70]) agent system has been chosen for the following reasons: *IMPACT* is an agent framework which allows for existing legacy code and data sources to be "agentized". Moreover, the behavior of an *IMPACT* agent, i.e., which actions it takes upon a state change, is specified declaratively by a set of rules. The possibility to employ existing implementations, together with the possibility of declarative agent specifications, makes the utilization of the declarative logic-programming components developed in this thesis within *IMPACT* agents straightforward.

Figure 2.1 shows the overall architecture of an *IMPACT* agent. All *IMPACT* agents have the same architecture, and hence the same components, but the *contents* of these components can be different, leading to different behaviors and capabilities offered by different agents.

Basically, the behavior of the agent is driven by an *action policy*. Each agent possesses a message box, which contains incoming and outgoing messages. On the basis of the content of the message box and of queries to legacy data (performed by means of function calls which abstract both from the structure of the underlying data and of the information sources), the actions to be performed are selected under a declarative semantics. Constraints ensure security and integrity of data and behavior. Actions may themselves be changes to available data, postings of messages to other agents, and so on.

2.2.1 Agent data structures

Agents are built "on top" of some existing body of code. Thus, to every agent, a set of types can be assigned, which contains all data types or data structures that the agent manipulates. As

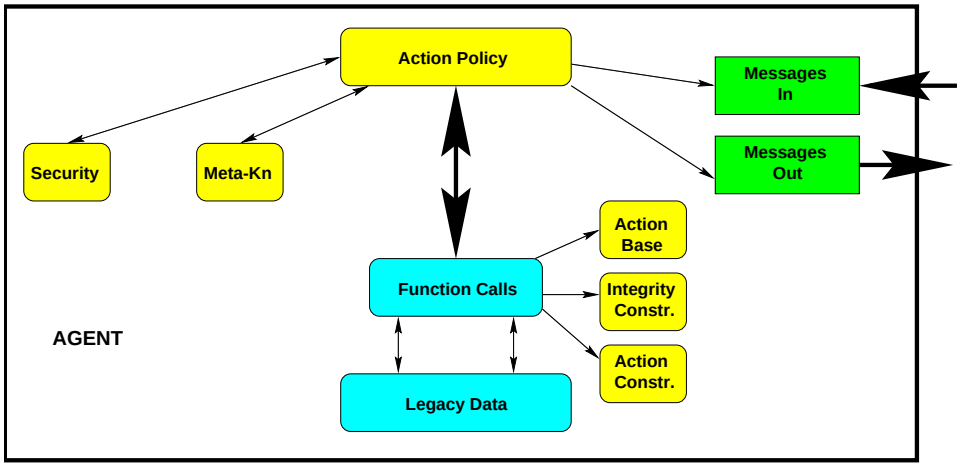

Figure 2.1: *IMPACT* agent architecture.

usual, each data type has an associated *domain* which is the space of objects of that type. The set of data structures is manipulated by a set of functions that are callable by external programs via *code calls*. Such functions constitute the *application programmer interface* (*API*) of the package on top of which the agent is built. An agent includes a specification of all signatures of these API function calls (i.e., types of the inputs to such function calls and types of the output of such function calls).

Every code call $\mathcal{S} : \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$, where $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are *terms*, i.e., either values or variables, is based on a body of software code, $\mathcal{S}$ (a so-called *software package*). Such a code call says "execute function $\mathtt{f}$ as defined in package $\mathcal{S}$ on the list of arguments". A code call can be evaluated providing it is ground, i.e., all arguments $\mathtt{t}_i$ must be values. Its output is a set of objects.

Code call atoms are expressions of the form $\mathtt{in}(\mathtt{t}, \mathtt{cc})$ or $\mathtt{notin}(\mathtt{t}, \mathtt{cc})$, where $\mathtt{t}$ is a term and $\mathtt{cc}$ is a code call. A ground term $\mathtt{t}$ succeeds (i.e., has answer *true*) if $\mathtt{t}$ is in the set of values returned by $\mathtt{cc}$, otherwise it fails (i.e., has answer *false*). If $\mathtt{t}$ is a variable, then a code call atom returns each value from the result of $\mathtt{cc}$, i.e., its answer is the set of ground substitutions for $\mathtt{t}$ such that the code call atom succeeds.

A *code call condition* is a conjunction of code call atoms and *constraint atoms*, which may involve deconstruction operations. An example of a constraint atom is $\mathtt{X} > 25$, where $\mathtt{X}$ is a variable. A code call condition checks whether the stated condition is true. In general, constraint atoms are of the form $\mathtt{t}_1 \circ \mathtt{t}_2$ where $\circ \in \{=, \neq, <, \leq, >, \geq\}$ and $\mathtt{t}_1, \mathtt{t}_2$ are terms.

Each agent has access to a message box data structure, together with some API function calls to access it.

At any given point in time, the actual set of objects in the data structures (and the message box) managed by the agent constitutes the *state* of the agent. The set of ground code calls which are true in it are identified as the state, $\mathcal{O}$, of the agent.

2.2.2 Actions

The agent has a set of *actions*. For example, reading a message from the message box, executing a request, updating the agent data structures, or even doing nothing is an action. Expressions $\alpha(\overline{t})$, where α is an action and $\overline{t}$ is a list of terms, are *action atoms*. They represent the sets of (ground) actions which result if all variables in $\overline{t}$ are instantiated by values. Only such actions may be executed by an agent. Every action has a precondition, $Pre(\alpha)$, a set of effects that describe how the agent state changes when the action is executed, and an *execution script* or *method* consisting of a body of physical code that implements the action.

2.2.3 Agent programs

Each agent has a set of rules (*action rules*) called the *agent program* specifying the principles under which the agent is operating. These rules specify, using deontic modalities, what the agent may do, must do, may not do, etc. Expressions $\mathbf{O}\alpha(\overline{t})$, $\mathbf{P}\alpha(\overline{t})$, $\mathbf{F}\alpha(\overline{t})$, $\mathbf{Do}\,\alpha(\overline{t})$, and $\mathbf{W}\alpha(\overline{t})$, where $\alpha(\overline{t})$ is an action atom, are called *action status atoms*. These action status atoms are respectively read as $\alpha(\overline{t})$ is *obligatory*, $\alpha(\overline{t})$ is *permitted*, $\alpha(\overline{t})$ is *forbidden*, *do* $\alpha(\overline{t})$, and the obligation to do $\alpha(\overline{t})$ is *waived*.

If A is an action status atom, then A and $\neg A$ are called *action status literals*. An *agent program* $\mathcal{P}$ is a finite set of rules of the form

$$A \quad \leftarrow \quad \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n,$$

where A is an action status atom, χ is a code call condition, and $L_1, \ldots, L_n$ are action status literals.

2.2.4 Semantics

Each agent program has a *formal semantics* which is defined in terms of semantical structures called *status sets*, i.e., sets of ground action status atoms. More specifically, the semantics of an agent is defined with respect to *feasible status sets*, which satisfy various conditions. First, a feasible status set S is required to be closed under the rules of the agent program and comply to the deontic and action axioms listed below. Second, concurrently executing all actions α such that $\mathbf{Do}\,(\alpha) \in S$ should take the agent from its current state $\mathcal{O}$ to another state $\mathcal{O}'$ which satisfies the *integrity constraints*, $\mathcal{IC}$, associated with the agent. Third, concurrent execution of the set of all actions α such that $\mathbf{Do}\,(\alpha) \in S$ should not violate any of the *action constraints*, $\mathcal{AC}$, associated with the agent.

The deontic and action axioms a feasible status set S has to obey for any ground action α, consist of *deontic and action consistency axioms*:

- *If* $\mathbf{O}\alpha \in S$, *then* $\mathbf{W}\alpha \notin S$,
- *If* $\mathbf{P}\alpha \in S$, *then* $\mathbf{F}\alpha \notin S$,

- *If* $\mathbf{P}\alpha \in S$*, then* $\mathcal{O}_{\mathcal{S}} \models Pre(\alpha)$ *(i.e.,* α *is executable in the state* $\mathcal{O}_{\mathcal{S}}$*),*

and of *deontic and action closure rules*:

- $\mathbf{O}\alpha \in S \rightarrow \mathbf{P}\alpha \in S$,
- $\mathbf{O}\alpha \in S \rightarrow \mathbf{Do}\,\alpha \in S$, and
- $\mathbf{Do}\,\alpha \in S \rightarrow \mathbf{P}\alpha \in S$.

Additionally, stronger semantical notions than feasible status sets have been introduced for *IMPACT* agents, namely *rational status sets* and *reasonable status sets*. Informally, rational status sets are minimal feasible status sets, i.e., no subset of the action status set can be removed while the program rules and deontic axioms are still satisfied. Reasonable status sets further restrict rational status sets by the treatment of negation in the spirit of the treatment of default negation for logic programs as introduced in the previous section. Thus, reasonable status sets extend the stable model semantics of logic programs as shown in [128]. Moreover, reasonable status sets have elegant important properties and are computationally not as hard as the other semantics, cf. [128].

There are also further components of *IMPACT* agents which are not relevant for our purposes here. A more detailed description of the *IMPACT* system and the semantics of *IMPACT* agents can be found in [128, 69].

3 Update Agents

"*A wise man changes his mind, a fool never will.*" (Spanish Proverb)

Logic programming has not only been conceived as a computational logic paradigm for problem solving – offering a number of advantages over conventional programming languages – and as a well-suited tool for declarative knowledge representation and common-sense reasoning [21] – possessing a high potential as a key technology to equip software agents with advanced reasoning capabilities in order to make those agents behave intelligently (e.g., [122]) – but it has also been realized that further work is needed on extending the current methods and techniques to fully support the needs of agents.

An important aspect is that an agent is situated in an environment which is subject to change. This requests the agent to adapt over time, and to adjust its decision making. For agents utilizing logic programming techniques for representing knowledge, this requires the agent to be capable of updating logic programs accordingly, in order to ensure adaptivity.

In a simple (but, as for currently deployed agent systems, realistic) setting, an agent's knowledge base, KB, may be modeled as a logic program, which the agent may evaluate to answer queries that arise. Given various approaches to semantics, the problem of evaluating a logic program is quite well-understood. However, an agent might be prompted to adjust its knowledge base KB after receiving new information in terms of an *update* U, given by a clause or a set of clauses that need to be incorporated into KB. Simply adding the rules of U to KB does not give a satisfactory solution in practice, and will result in inconsistency even in simple cases. For example, if KB contains the rules $a \leftarrow b$ and $b \leftarrow$, and U consists of the rule $\neg a \leftarrow$ stating that a is false, then the union $KB \cup U$ is not consistent under predominant semantics such as the answer set semantics [85] or the (extended) well-founded semantics [129].

Besides an underlying update semantics, which specifies how new, possibly inconsistent information is to be incorporated into the knowledge base, an agent needs to have a certain *update policy*, i.e., a specification of how to react upon the arrival of an update. For instance, the policy may specify the change or retraction of certain rules from the knowledge base, given some particular information. More precisely, given a new piece of information, the update mechanism has to address the following questions:

1. Which facts and rules should be incorporated?
2. How should facts and rules be incorporated?