Prospective Logic Programming with ACORDA

Gonçalo Lopes and Luís Moniz Pereira
Centro de Inteligência Artificial - CENTRIA
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
`goncaloclopes@gmail.com`
`lmp@di.fct.unl.pt`

Abstract

As we face the real possibility of modelling programs that are capable of non-deterministic self-evolution, we are confronted with the problem of having several different possible futures for a single such program. It is desirable that such a system be somehow able to *look ahead*, prospectively, into such possible futures, in order to determine the best courses of evolution from its own present, and then to prefer amongst them. This is the objective of the ACORDA, a prospective logic programming system. We start from a real-life working example of differential medical diagnosis illustrating the benefits of addressing these concerns, and follow with a brief description of the concepts and research results supporting ACORDA, and on to their implementation. Then we proceed to fully specify the implemented system and how we addressed each of the enounced challenges. Next, we take on the proffered example, as codified into the system, and describe the behaviour of ACORDA as we carefully detail the resulting steps involved. Finally, we elaborate upon several considerations regarding the current limitations of the system, and conclude with the examination of possibilities for future work.

1 Introduction

Part of the Artificial Intelligence community has struggled, for some time now, to make viable the proposition of turning logic into an effective programming language, allowing it to be used as a system and application specification language which is not only executable, but on top of which one can demonstrate properties and proofs of correctness that validate the very self-describing programs which are produced. At the same time, AI has developed logic beyond the confines of monotonic cumulativity and into the non-monotonic realms that are typical of the real world of incomplete, contradictory, arguable, revised, distributed and evolving knowledge. Over the years, enormous amount of work and results have been achieved on separate topics in logic programming (LP) language semantics, belief revision, preferences, and evolving programs with updates[DP07, PP05, ABLP02].

As we now have the real possibility of modelling programs that are capable of non-deterministic self-evolution, through self-updating, we are confronted with the problem of having several different possible futures for a single starting program. It is desirable that such a system be able to somehow *look ahead* into such possible futures to determine the best paths of evolution from its present, at any moment. This involves a notion of simulation, in which the program is capable of conjuring up hypothetical *what-if* scenarios and formulating abductive explanations for both external and internal observations. Since we have multiple possible scenarios to choose from, we need some form of preference specification, which can be either a priori or a posteriori. A priori preferences are embedded in the program's own knowledge representation theory and can be used to produce the most relevant hypothetical abductions for a given state and observations, in order to conjecture possible future states. A posteriori preferences represent choice mechanisms, which enable the program to commit to one of the hypothetical scenarios engendered by the relevant abductive theories. These mechanisms may trigger additional simulations in order to posit which new information to acquire, so more informed choices can be enacted, in particular by restricting and committing to some of the abductive explanations along the way.

A large class of higher-order problems and system specifications exist which would benefit greatly from this strategy for evolving knowledge representation, especially in domains closer to the human level of reasoning such as expert systems for medical diagnosis, and systems for enhanced proactive behavioural control.

1.1 Iterated Differential Diagnosis

In medicine, differential diagnosis is the systematic method physicians use to identify the disease causing a patient's symptoms. Before a medical condition can be treated it must first be correctly diagnosed. The physician begins by observing the patient's symptoms, taking into consideration the patient's personal and family medical history and performing additional examinations if current information is lacking. Then the physician lists the most likely causes. The physician asks questions and performs tests to eliminate possibilities until he or she is satisfied that the single most likely cause has been identified. The term differential diagnosis also refers to medical information specially organized to aid in diagnosis, particularly a list of the most common causes of a given symptom, annotated with advice on how to narrow down the list.

This listing of the most likely causes is clearly an abduction process, supported by initial observations. Encoding this process in logic programming would result in a set of literals and rules representing both the expected causes and the knowledge representation theory leading up to them. This set could be subsequently refined through a method of prospection with a priori and a posteriori preferences, like the one described above. This prospective process can result in the decision to perform additional observations on the external environment, and be iterated.

In this case, the external observations correspond to the disease's signs and other examinations performed by the physician during the process of diagnosis. The abductive explanations correspond to the possible causes (e.g. medical conditions or simply bad habits) responsible for the symptoms. A priori preferences and expectations are necessarily determined according to the patient's symptoms and they can be updated during the course of the prospection mechanism. An appropriate knowledge representation theory can be derived from current medical knowledge about the causes of a given disease.

Being such a good example of abduction, we provide next a practical example of iterated differential diagnosis, based upon real medical knowledge from the field of dentistry, in order to better illustrate the kinds of problems involved. Further on we shall proffer an encoding of the problem on our system, along with results from an interactive diagnosis session.

1.1.1 Differential Diagnosis in Dentistry: A Use Case

A patient shows up at the dentist with signs of pain upon teeth percussion. The expected causes for the observed signs are:

- Periapical lesion (endodontic or periodontal source)
- Horizontal Fracture of the root and/or crown
- Vertical Fracture of the root and/or crown

Several additional examinations can be conducted to determine the exact cause, namely:

- X-Ray for determination of radiolucency or fracture traces
- X-Ray for determination of periapical lesion source
- Check for tooth mobility
- Measurement of gingival pockets

Aside from presenting multiple hypotheses for diagnosis, the knowledge exhibited by the practitioner must necessarily evolve in time, as he or she performs relevant examinations which will attempt to disqualify all but one of the possible explanations. Current examinations depend on knowledge acquired in the past, which, in turn, will end up influencing the observations and inferences which will be drawn in the future. Developing a system that is capable of modelling the evolution of a program in order to draw such inferences is a demanding but obviously useful challenge.

1.2 Prospective Logic Programming

This problem is already at hand, since there are working implementations of a logic programming language which was specifically designed to model such program evolutions. EVOLP [ABLP02] is a non-deterministic LP language, having a well-defined semantics that accounts for self-updates, and we intend to use it to model autonomous agents capable of evolving by making proactive commitments concerning their imagined prospective futures. Such futures can also be prospections of actions, either from the outside environment or originating in the agent itself. This implies the existence of partial internal and external models, which can already be modelled and codified with logic programming.

It is important to distinguish this mechanism of prospection from your average planning problem, in which we search a state space for the means to achieve a certain goal. In planning, this search space is inherently fixed and implicit, by means of the set of possible actions that we can use to search for the goal. In future preference, we are precisely establishing that search space, defining our horizon of search, which isn't a priori known and can even change dynamically with the very process of searching. It intends to be a pre-selection prior to planning, but essentially distinct from planning itself.

A full-blown theory of preferences[AP00] has also been developed in LP along with EVOLP, as well as an abduction theory [APS04] and several different working semantics for just such non-monotonic reasoning, such as the Stable Models [GL88] and Well-Founded Semantics [vGRS91]. All of these have thoroughly tested working implementations. The problem is then how to effectively combine these diverse ingredients in order to avail ourselves of the wherewithal to solve the problems described above. We are now ready to begin addressing such general issues with the tools already at hand, unifying several different research results into powerful implementations of systems exhibiting promising new computational properties. This is the main objective of the ACORDA system[1].

First, we provide a brief description of the concepts and previous research results supporting ACORDA, and their implementations[2]. Then we proceed to fully specify the implemented system and how we have addressed each of the enounced challenges. Next, we take on the real-life working example of differential medical diagnosis, showing how it can be codified into the system, and describing the behaviour of ACORDA, as we carefully detail the resulting steps. Finally, we elaborate upon several considerations regarding the current limitations of the system and conclude examining possibilities for future work.

2 Logic Programming Framework

2.1 XSB-XASP Interface

The Prolog language has been for quite some time one of the most accepted means to codify and execute logic programs, and as such has become a useful tool for research and application development in logic programming. Several stable implementations have been developed and refined over the years, with plenty of working solutions to pragmatic issues ranging from efficiency and portability to explorations of language extensions. The XSB Prolog system is one of the most sophisticated, powerful, efficient and versatile among these implementations, with a focus on execution efficiency and interaction with external systems, implementing program evaluation following the Well-Founded Semantics (WFS) for normal logic programs.

Two of its hallmark characteristics make it a particularly useful system on top of which to implement ACORDA, and many of its supporting subsystems. First of all, the tabling mechanism [Swi99], in which the results of particular queries are stored for later reuse, can provide not only an enormous decrease in time complexity, but also allow for solutions to well-known problems in the LP community, such as query loop detection.

Secondly, its aiming for external systems interaction eventually resulted in the development of an interface to Smodels [NS97], one of the most successful implementations of the Stable Models semantics over generalized logic programs, also known as the Answer

[1]ACORDA means literally “wake-up” in Portuguese. The *ACORDA* system project page is temporarily set up at: `http://articaserv.ath.cx/`

[2]Working implementations of *Dynamic Logic Programming*, *EVOLP* and *Updates plus Preferences using DLP* available online at: `http://centria.di.fct.unl.pt/~jja/updates`

Set semantics. The SM semantics has become the cornerstone for the definition of some of the most important results in logic programming of the past decade, providing an increase in logic program declarativity and a new paradigm for program evaluation. Many of the ACORDA subsystems are defined on top of the Stable Models (SM) semantics, and as such, this integration proves extremely useful, and even accounts for new and desirable computational properties that neither of the systems could provide on its own.

The XASP interface [CSW] (standing for XSB Answer Set Programming) provides two distinct methods of accessing Smodels[3]. The first one is for using Smodels to obtain the stable models of the so-called residual program, the one that results from a query evaluated in XSB using tabling. This residual program is represented by delay lists, that is, the set of undefined literals for which the program could not find a complete proof, due to mutual dependencies or loops over default negation for that set of literals, which are detected by the XSB tabling mechanism. This method allows us to obtain any two-valued semantics in completion to the three-valued semantics the XSB system provides. The second method is to build up a clause store, adding rules and facts to compose a generalized logic program that is then parsed and sent to Smodels for evaluation, thereafter providing access to the computed stable models back to the XSB system.

This kind of integration allows one to maintain the relevance property for queries over our programs, something that the Stable Models semantics does not originally enjoy. In Stable Models, by the very definition of the semantics, it is necessary to compute all the models for the whole program. In our system, we sidestep this issue, using XASP to compute the relevant residual program on demand, usually after some degree of transformation. Only the resulting program is then sent to Smodels for computation of possible futures. We believe that such system integrations are crucial in order to extend the applicability of the more refined and declarative semantics that have been developed in the field of AI.

2.2 Evolving Logic Programs

Modelling the dynamics of knowledge changing over time has been an important challenge for LP. Accounting for the specification of a program's own evolution is essential for a wide variety of modern applications, and necessary if one is to model the dynamics of real world knowledge. Several efforts were conducted in lieu of developing a unified language that could be both expressive and simple, following the spirit of declarative programming that is characteristic of LP. The language EVOLP [ABLP02] is one of the most powerful results from this research area with working implementations.

EVOLP generalizes LP in order to provide a general formulation of logic program updating, by permitting rules to indicate assertive conclusions having the form of program rules. Such assertions, whenever they belong to a model of the program P, can be employed to generate an updated version of P. This process can then be iterated on the basis of the new program. When the program semantics affords several program models, branching evolution will occur and several evolution sequences are possible. The ability of EVOLP to nest rule assertions within assertions allows rule updates to be themselves updated over time, conditional on each evolution strand. The ability to include assertive

[3]The XSB Logic Programming system and Smodels are freely available at: `http://xsb.sourceforge.net` and `http://www.tcs.hut.fi/Software/smodels`

literals in rule bodies allows for looking ahead on program changes and acting on that knowledge before the changes actually take place.

2.2.1 Self-evolving Logic Programs

The approach used to define EVOLP aimed for minimality in regard to classic logic programs. The objective was to identify the necessary conditions allowing the new capabilities of evolution and updating, and then minimally adding constructs to LP in order to account for them. Since we are envisaging updates to logic programs, it is necessary to provide a means to state that, under some conditions, some rule is to be added to the program. However, to allow for the non-monotonicity of rules, it is also necessary to provide some sort of negation in rule heads, so as to let instances of older rules be supervened by more recent rules updating them. Under these minimal conditions it is possible to express any kind of logic program evolution and, as such, EVOLP aimed to satisfy both.

The negation in rule heads is provided by the very definition of generalized LPs, while the statement of updates can be specified by augmenting this language with the reserved predicate *assert/1*, whether as the rule head or in its body. The sole argument of this predicate is itself a full blown rule in order to account for the possibility of arbitrary nesting. The formal inductive definition of the EVOLP language can be presented thus:

Definition 1[ABLP02]. *Let $\mathcal{L}$ be any propositional language (not containing the predicate assert/1). The extended language $\mathcal{L}_{assert}$ is defined inductively as follows:*

1. *All propositional atoms in $\mathcal{L}$ are propositional atoms in $\mathcal{L}_{assert}$.*
2. *If each of $L_0, \ldots, L_n$ is a literal in $\mathcal{L}_{assert}$ (i.e. a propositional atom A or its default negation not A), then $L_0 \leftarrow L_1, \ldots, L_n$ is a generalized logic program rule over $\mathcal{L}_{assert}$.*
3. *If R is a rule over $\mathcal{L}_{assert}$ then assert(R) is a propositional atom of $\mathcal{L}_{assert}$.*
4. *Nothing else is a propositional atom in $\mathcal{L}_{assert}$.*

An evolving logic program over a language $\mathcal{L}$ is a (possibly infinite) set of generalized logic program rules over $\mathcal{L}_{assert}$.

It should be noted that the basic syntax provides no explicit retract construct because in fact it has no need of one. Retraction of rules can be encoded in EVOLP simply by allowing for default negation to appear in rule heads.

The semantics of such self-evolving logic programs is provided by a set of *evolution stable models*, each of which is a sequence of interpretations or states. Each evolution stable model describes some possible self-evolution of one initial program after a given number n of evolution steps. Each self-evolution is represented by a sequence of programs, each program corresponding to a state.

These sequences of programs are treated as in Dynamic Logic Programs [ALP$^+$00], where the most recent rules are put in force, and previous rule instances are valid by inertia insofar as possible, as long as they do not conflict with more recent ones.

2.3 Preferential Theory Revision

The application of preferential reasoning over logic programs in the form of preferences between rules has been successfully attempted, including combinations of such rule preferences with program updates[AP00], and the updating of preferences themselves. However, a crucial ingredient had been missing, that of considering the possible abductive extensions to a theory, as expressed by means of a logic program and the integration in it of preferences over such extensions.

Abduction plays a crucial role in belief revision and diagnosis, and also in the development of hypotheses to explain some set of observations, a common consequence of working under the scientific method. Such abductive extensions to a theory can be expressed by sets of *abducibles*, over which we should be able to express conditional priority relations. Abducibles may be thought of as the hypothetical solutions or possible explanations that are available for conditional proof of a given query.

This ability of construing plausible extensions to one's theory is also vital for logic program evolution, so that the program is capable of self-revision and theorizing, providing new and powerful ways on which it can guide the evolution of its knowledge base, by revising incorrect or incomplete behaviour.

Such preferential theory revision has been approached in [DP05, DP07] and will be presented next as part of the logic programming framework of the ACORDA prospective system.

2.3.1 Language

Let $\mathcal{L}$ be a first order language. A domain literal in $\mathcal{L}$ is a domain atom A or its default negation *not* A, the latter expressing that the atom is false by default (CWA). A domain rule in $\mathcal{L}$ is a rule of the form:

$$A \leftarrow L_1, \ldots, L_t \quad (t \geq 0)$$

where A is a domain atom and $L_1, \ldots, L_t$ are domain literals. The following convention is used. Given a rule r of the form $L_0 \leftarrow L_1, \ldots, L_t$, we write $H(r)$ to indicate L_0 and $B(r)$ to indicate the conjunction $L_1, \ldots, L_t$. We write $B^+(r)$ to indicate the conjunction of all positive literals in $B(r)$, and $B^-(r)$ to indicate the conjunction of all negated literals in $B(r)$. When $t = 0$ we write the rule r simply as L_0. Let $\mathcal{A} \subseteq \mathcal{L}$ be a set of domain atoms, the *abducibles*.

2.3.2 Preferring Abducibles

To express preference criteria among abducibles, we introduce the language $\mathcal{L}^*$. A relevance atom is one of the form $a \lhd b$, where a and b are abducibles. $a \lhd b$ means that the abducible a is more relevant than the abducible b. A relevance rule is one of the form:

$$a \lhd b \leftarrow L_1, \ldots, L_t \quad (t \geq 0)$$

where $a \lhd b$ is a relevance atom and every $L_i (1 \leq i \leq t)$ is a domain literal or a relevance literal. Let $\mathcal{L}^*$ be a language consisting of domain rules and relevance rules.

The original definition of *abductive stable models* given in [DP07] considered logic programs already infused with preferences between rules. The framework considered in this work does not assume this, and as such the transformation of programs over $\mathcal{L}^*$ is slightly different, as it considers only generalized LPs.

The following definition provides us with the syntactical transformation from the language $\mathcal{L}^*$ to normal logic programs, with the abducibles being codified as simultaneous even-loops that guarantee mutual exclusion, i.e. that only one abducible is present in each model. Relevance rules are also codified so as to defeat the abducibles which are less preferred when the body of the rule is satisfied.

Definition 2[DP05, DP07]. *Let Q be a program over $\mathcal{L}^*$ with set of abducibles $\mathcal{A}_Q = \{a_1, \ldots, a_m\}$. The program $P = \Sigma(Q)$ with abducibles $\mathcal{A}_P = \{abduce\}$ is obtained as follows:*

1. *P contains all the domain rules in Q*
2. *for every $a_i \in \mathcal{A}_Q$, P contains the domain rule:*
   $confirm(a_i) \leftarrow expect(a_i), not\ expect\_not(a_i)$
3. *for every $a_i \in \mathcal{A}_Q$, P contains the domain rule:*
   $$\begin{aligned} a_i \leftarrow\ & abduce, not\ a_1, \ldots, not\ a_{i-1}, not\ a_{i+1}, \ldots, not\ a_m, \\ & confirm(a_i), not\ neg\_a_i \end{aligned}$$
4. *for every relevance rule r in Q, P contains a set of domain rules obtained from r by replacing every relevance atom $x \lhd y$ in r with the following domain rule:*

   $neg_y \leftarrow L_0, \ldots, L_n, B_x^+, B_y, not\ neg_x$

3 ACORDA Architecture and Implementation

The basis for the ACORDA architecture is a working implementation of EVOLP on which we can evaluate for truth literals following both a three-valued or a two-valued logic. Since we aim for autonomous abduction processes that are triggered by constraints or observations, from within the knowledge state of an agent, we need a means to express how this triggering is accomplished. In our system, we resort to the notion of *observable* in order to model and specify this kind of behaviour.

An observable is a quaternary relation amongst the *observer*, i.e. that which is performing the observation; the *observed*, that which is the target of the observation; the result of the *observation* itself; and the *truth value* associated with the observation. This relation is valid both to express program based self-triggering of internal queries, as well as observations requested from an exterior oracle to the program, and from the program directly to the environment. For example, the observable

```
observable(prog,prog,Query,true)
```

represents an observation in which the observer is the program, that which is observed is also the program, the observation is the specified Query and the positive truth value means the observation must be proven true. In this case, we expect that such an observable, by becoming active, triggers the self-evaluation of Query in the current knowledge state, in the process resorting to any relevant and expected abducibles that can account for the observation.

As a consequence of the fact that only one of the relevant abducibles can effectively be chosen, corresponding to some possible world model, we may have several different results for explaining such self-observations. These results represent the possible knowledge futures that the program can infer from the current knowledge state towards explaining a given hypothetical observation, and also the choices that the program can make in order to evolve towards a new state where the observation holds. To make the correct choice, we implement a priori and a posteriori prospective systems of preferences. After this prospection takes place, the system is updated with the relevant literals that explain the observation, and the transition to the next knowledge state is then complete. This iterative process can be repeated for as long as we like, producing autonomous evolution of the program, interleaved with external updates that can reach the system independently, in between every such iteration. For simplicity, we presently assume that external updates do not reach the system during each abductive process. The steps in an ACORDA evolution loop are detailed below, and we proceed to explain them in detail in the following sections.

1. Determine the active program-to-program generated observations for the current state and their respective queries.

2. Determine the expected abducibles which are confirmed relevant to support the active observations and which do not violate the integrity constraints.

3. Compute the abductive stable models generated by virtue of those confirmed abducibles, considering the currently active a priori preferences amongst them.

4. If necessary, activate a posteriori choice mechanisms to possibly defeat some of the resulting abductive stable models.

5. Update the current knowledge state with the chosen abducible that can effectively satisfy the activated observations and integrity constraints.

3.1 Active Program-to-Program Generated Observations

The active self-observations for a given state are obtained from a special kind of clause which indicates that certain program-to-program observations should be made, in exclusion of all others. These *on_observable/4* clauses follow the same structure of the *observable/4* clauses, but they do not represent the observations themselves. They point rather to the self-observations that the ACORDA system should perform in the next step of evolution. In order to ease codification in the ACORDA system, we consider additional *observable/3* and *on_observable/3* clauses, which default the truth value to `true`.

Those *on_observable/4* clauses whose body holds true in the Well-Founded Model (WFM), the three-valued logic model derived from the current knowledge state, will trigger an internal observation expressed by their conclusion. The reason why we opt for this skeptical semantics in this step is that we do not wish to allow just any kind of abduction to determine such activated observations. Since we codify abduction using even-loops over default negation, we would risk having those even-loops generate several possible models for active observations if we considered the Stable Models semantics. With the Well-Founded Semantics, we guarantee that a single model of active observations exists, composed of those literals which can be proven true in the WFM, with any literals supported by abductions being marked as undefined. A much more expensive alternative, of course, would be to find those observations true in the intersection of all abductive Stable Models.

Also, in a knowledge base of considerable dimension, the number of active program-to-program observations for a given knowledge state can be huge. Typically, we want to give attention to only a subset of the observations that are active. There are multiple ways in which this subset could be determined, either by using utility functions, additional preference relations or priorities, or just any other means of determining the most important observables. In our system, we leave this function to be determined by the user, since it will hinge much on the kind of application being developed. By default, the system attends to all the observations that are active.

3.2 Expected and Confirmed Abducibles

Each active program-to-program observation will be launched as part of a conjunctive query, which the system will attempt to satisfy by relying on any expected abducibles which become confirmed by not otherwise being defeated, along with enforcing satisfaction of integrity constraints. Since only a single abducible can currently be derived from the even loops expressing alternative abductions, it follows that this abducible must be able to satisfy the conjunction of active observations and attending integrity constraints in order for evolution to succeed. This notion can of course be extended to support abducible sets, and in that case one may want to consider the minimal sets of abducibles that satisfy the conjunction. One way to allow for sets is to attach a single abducible to each desired alternative set. This is currently outside the scope and essence of the project, but will be considered in future developments, as mentioned in section 5.

Each abducible needs first to be expected (i.e. made available) by a model of the current knowledge state. Next it needs to be confirmed in that model before it can be used to satisfy or explain any kind of prospective observation. This is achieved via the *expect/1* and *expect_not/1* clauses, which indicate conditions under which an expectable abducible is indeed expected and confirmed for an observation given the current knowledge state. A relevant expected and confirmed abducible must be present in the current derivation tree for the conjunction of all observations, satisfy at least one *expect/1* clause and none of its *expect_not/1* clauses. It must also be coded over an even-loop using the system's general *abduce/1* clause, detailed below along with the general *confirm/1* clause encoding the requisite properties of an expected and confirmed abducible.

```
      confirm(X)  <- expect(X), not expect_not(X), abduce(X).
       abduce(X)  <- not abduce_not(X).
   abduce_not(X)  <- abduce(X).
```

The latter two clauses are defined for every abducible by instantiating its variable to it, so as to produce an even-loop with the *abduce_not/1* clause, and guarantee that the confirmed abducibles come up undefined in the WFM of the program, and hence in the residual program computed by the XSB Prolog-based EVOLP meta-interpreter as explained in section 2.1. This residual program is a set of cyclic strongly connected graphs of the interdependencies among the undefined atoms in the Well-Founded Model of a program, without the presence of any true or false literals, for these have already been partially evaluated[DMS].

3.3 Computation of the Abductive Stable Models

Determination of relevant abducibles can be performed by examination of the residual program for ground literals which are arguments to *confirm/1* clauses. Currently we need to assume ground literals since we will be producing a program transformation over the residual program based on the set of confirmed abducibles, which will be sent directly to Smodels for computation of the hypothetical confirmed abducibles' generated scenarios. Smodels needs all literals to be instantiated, or have an associated domain for instantiation. This limitation is present in many other state-of-the-art logic programming systems and its solution is not the main point of this work. Nevertheless, it is worth considering that XSB's unification algorithm can ground some of the variables automatically during the top-down derivation of observations hinging on abducibles.

Once the set of relevant confirmed abducibles is determined from the program's current knowledge state, all that remains before applying the preference transformation described in section 2.3 is to determine the active a priori preferences that are relevant for that set. This is merely a query for all preference literals whose heads indicate a preference between two abducibles that belong to the set, and whose body is true in the Well-Founded Model of the current knowledge state.

The XASP package[CSW] allows the programmer to collect rules in an XSB clause store. When the programmer has determined that enough clauses have been added to the store to form a semantically complete sub-program, the program is then *committed*. This means that information in the clauses is copied to Smodels, codified using Smodels data structures so that stable models of those clauses can be computed and examined.

When both the relevant abducibles and the active preferences are determined, the transformation is applied, and every resulting clause is sent to the XASP store, which is reset beforehand in preparation for the stable models computation. The transformed residual program is then committed to Smodels and we make use of the XASP interface to obtain back the literals corresponding to abducibles from all the resulting stable models of the transformed program. These models form the Abductive Stable Models of the current knowledge state given the active observations and available relevant abducibles.

3.4 A Posteriori Choice Mechanisms

If everything goes well and only a single model emerges from computation of the abductive stable models, the ACORDA cycle terminates, and the resulting abducible is updated into the next state of the knowledge base. In most cases, however, we cannot guarantee the emergence of a single model, since the active preferences may not be sufficient to defeat enough abducibles. In these situations, the ACORDA system has to resort on additional information for making further choices, but supported on what?

A given abducible can be defeated in any one of two cases: either by satisfaction of an *expect_not/1* clause for that abducible, or by satisfaction of a preference rule that prefers another abducible instead. However, the current knowledge state may be insufficient to satisfy any of these cases for all abducibles except one, or else a single model would have already been abduced. It is then necessary that the system obtain the answers it needs from somewhere else, namely from making experiments on the environment or from querying an outside entity.

An agent or program that is completely isolated from its environment is limited in the kinds of reasoning it can produce. If we are to model agents which can face up proactively to the problems of the real world, they need to have the ability to probe the outside environment in order to make experiments which permit it to enact a more informed choice. We next consider the mechanism by means of which the program can pose questions to external systems, be they other agents, actuators, sensors or other procedures. Each of these serves the purpose of an *oracle*, which the program can probe with observations of its own. These observations are of the form

```
observable(prog,Oracle,Query,Value)
```

representing that the program is performing the observation Query on the specified Oracle. These clauses can normally only be satisfied if the system has activated oracle probing, and if the conditions are met for using the prescribed oracle. In some situations, not all oracles may be available for the agent to query.

ACORDA consequently activates its a posteriori choice mechanisms, which consist in attempting to satisfy additional self-observations of the top-down conjunctive query taking into account the possible satisfaction of more *expect_not/1* clauses or preference rules. Each such observation is an ACORDA cycle in its own right, counting on enabled oracles' activation to perform external queries if necessary. Any such observation can spawn even more observations, as long as there are clauses to inspect and oracles to interrogate. Information gleaned from the oracles can produce numerous side-effects, besides the possible defeat of previous abducibles, namely: supporting new abductions; activating preference relations previously unaccounted for; or even lending support by themselves to the satisfaction of the original query.

The results from performed experiments are currently tabled, to allow for reuse of the retrieved information for the current knowledge state. We assume the environment does not change while the ACORDA system is performing its abductive reasoning. There are many situations in which this assumption would be unacceptable but, once again, these fall outside the immediate scope of the project. They are discussed as current system limitations in section 5.

Eventually, the ACORDA system will have defeated enough models that it can guarantee a single one will emerge with the additional information gathered, or that it has exhausted every means to make the choice. In any case, it will commit to the literals gathered from the observations, querying the user for the final choice, if indeed it fails to produce a single course of evolution by itself.

The finalizing committing update can, of course, trigger additional observations and turn the attention of the system to other matters, but always in a well-founded manner, so that the agent is capable of autonomous reasoning justified by its initial program and self-guided evolution. We believe that the ACORDA system is an interesting experiment on new forms of combined reasoning that deal with new emerging problems already beginning to present themselves. Moreover, even though an experimental system, it's already able to solve empirically concrete real-world problems of some degree of complexity with classical LP declarative programming style, as the next section will demonstrate.

4 Modelling Prospective Logic Programs

We will now resume the issue of codifying the example of differential diagnosis in dentistry presented in section 1.1.1 using the ACORDA system. The results obtained during an interactive diagnosis section are then detailed, along with annotations on the workings of the system.

The scenario can be intuitively modelled using the provided syntax for prospective logic programs on the ACORDA system, resulting in the initial program exhibited below. It should be noted that, as a matter of coding syntax, the relevance operator $\triangleleft$ is represented by the atom `<|`.

```
==== Initial Signs ====

on_observable(prog,prog,percussion_pain_cause) <- percussion_pain. (A)

==== First Phase of Differential Diagnosis ====

percussion_pain_cause <- periapical_lesion.
percussion_pain_cause <- vertical_fracture.
percussion_pain_cause <- horizontal_fracture.

periapical_lesion <- confirm(periapical_lesion).
expect(periapical_lesion) <- profound_caries.
expect(periapical_lesion) <- % expected in the context of an observation:
    on_observable(prog,prog,percussion_pain_cause).                (B)
expect_not(periapical_lesion) <- fracture_traces, not radiolucency.

vertical_fracture <- confirm(vertical_fracture).
expect(vertical_fracture) <- on_observable(prog,prog,percussion_pain_cause).
expect_not(vertical_fracture) <- radiolucency, not fracture_traces.

horizontal_fracture <- confirm(horizontal_fracture).
expect(horizontal_fracture) <- on_observable(prog,prog,percussion_pain_cause).
expect_not(horizontal_fracture) <- radiolucency, not fracture_traces.

periapical_lesion <| horizontal_fracture <-
    profound_caries, not percussion_pain.

periapical_lesion <| vertical_fracture <-
    profound_caries, not percussion_pain.

horizontal_fracture <| vertical_fracture <- low_mobility.
vertical_fracture <| horizontal_fracture <- high_mobility.
```

```
==== Second Phase of Differential Diagnosis ====

on_observable(prog,prog,periapical_lesion_source) <-                (C)
    periapical_lesion.

periapical_lesion_source <- endodontic_lesion.
periapical_lesion_source <- periodontal_lesion.

endodontic_lesion <- confirm(endodontic_lesion).
expect(endodontic_lesion) <-
    on_observable(prog,prog,periapical_lesion_source).
expect(endodontic_lesion) <- devitalization.
expect(endodontic_lesion) <- not gingival_pockets.
expect_not(endodontic_lesion) <-                                    (D)
    gingival_pockets, not devitalization.

periodontal_lesion <- confirm(periodontal_lesion).
expect(periodontal_lesion) <-
    on_observable(prog,prog,periapical_lesion_source).
expect(periodontal_lesion) <- devitalization.
expect(periodontal_lesion) <- gingival_pockets.
expect_not(periodontal_lesion) <- neg_gingival_pockets.

periodontal_lesion <| endodontic_lesion <- gingival_pockets.

==== Available Experiments ====

radiolucency <- observable(prog,xray,radiolucency).
fracture_traces <- observable(prog,xray,fracture_traces).          (E)
high_mobility <- observable(prog,mobility_check,high_mobility).
low_mobility <- observable(prog,mobility_check,low_mobility).
gingival_pockets <- observable(prog,pockets_check,gingival_pockets).
neg_gingival_pockets <-
    observable(prog,pockets_check,gingival_pockets,false).
devitalization <- observable(prog,periapical_xray,devitalization).

==== Available Oracles ====

observable(prog,xray,Q,S) <- oracle,prolog((oracleQuery(xray(Q),T),S = T)).
observable(prog,mobility_check,Q,S) <-
    oracle,prolog((oracleQuery(mobility_check(Q),T),S = T)).
observable(prog,pockets_check,Q,S) <-
    oracle,prolog((oracleQuery(pockets_check(Q),T),S = T)).
observable(prog,periapical_xray,Q,S) <-
    oracle,prolog((oracleQuery(periapical_xray(Q),T),S = T)).
```

4.1 Interactive Session Results with ACORDA

After the initial program is loaded into the ACORDA evolving knowledge base, we provide the patient's signs (i.e. `percussion_pain`) as an external update to the system.

Running a cycle of introspection over the ACORDA system produces the reasoning steps detailed below. The question marks are used to indicate queries to an external oracle, followed by the answer instances given by it.

1. `About to launch new introspection on selected active observables:`
   `[on_observable(prog,prog,percussion_pain_cause)]`
2. `Relevant abducibles for current introspection:`
   `[horizontal_fracture,periapical_lesion,vertical_fracture]`
3. `Partial models remaining after a priori preferences:`
   `[[horizontal_fracture],[periapical_lesion],[vertical_fracture]]`
4. `About to launch new introspection on selected active observables:`
   `[on_observable(prog,prog,percussion_pain_cause)]`
5. `Confirm observation: xray(fracture_traces)`
   `(true, false or unknown)? false.`
6. `Confirm observation: xray(radiolucency)`
   `(true, false or unknown)? true.`
7. `Relevant abducibles for current introspection:`
   `[periapical_lesion]`
8. `Partial models remaining after a priori preferences:`
   `[[periapical_lesion]]`

The event `percussion_pain` triggers the active observable `percussion_pain_cause` (cf. rule A) which is a program-to-program generated observation, that is, it will cause a top-down query to be launched to attempt satisfaction of the observable. The system goes down the derivation tree for `percussion_pain_cause`, encountering even-loops over default negation for each relevant abducible in the query, i.e. the literals that are output in 2. In this case, the expectations are triggered only in the context of the currently active observation (cf. rule B).

Afterwards, ACORDA launches queries for the a priori preferences which concern the relevant abducibles, which can themselves be dependent on abductive or observable loops. These abducibles and preferences determine the transformation specified in section 2.3. For the computed residual program and the set of abducibles in 2, we show part of the performed transformation, which is then placed in the XASP clause store:

```
percussion_pain_cause <- horizontal_fracture.
percussion_pain_cause <- periapical_lesion.
percussion_pain_cause <- vertical_fracture.
confirm(horizontal_fracture) <- not expect_not(horizontal_fracture).
confirm(periapical_lesion) <- not expect_not(periapical_lesion).
confirm(vertical_fracture) <- not expect_not(vertical_fracture).
expect_not(horizontal_fracture) <- radiolucency,not fracture_traces.
expect_not(periapical_lesion) <- fracture_traces,not radiolucency.
expect_not(vertical_fracture) <- radiolucency,not fracture_traces.

horizontal_fracture <-
    abduce, confirm(horizontal_fracture), not periapical_lesion,
    not vertical_fracture, not neg_horizontal_fracture.
```

```
false <- abduce, horizontal_fracture, neg_horizontal_fracture.

periapical_lesion <-
    abduce, confirm(periapical_lesion), not horizontal_fracture,
    not vertical_fracture, not neg_periapical_lesion.
false <- abduce, periapical_lesion, neg_periapical_lesion.

vertical_fracture <-
    abduce, confirm(vertical_fracture), not horizontal_fracture,
    not periapical_lesion, not neg_vertical_fracture.
false <- abduce,vertical_fracture,neg_vertical_fracture.
```

The 'false' literal is equivalently used here to model integrity constraints in our program, instead of the usual empty head that is adopted in Smodels' implementation. The potential observations which can satisfy some of the literals in the residual programs, namely the *expect_not/1* clauses are codified over even-loops, but are ommitted for clarity, since in the first phase of the system no external observations are allowed.

After the clauses are stored, the stable models of the processed residual program are computed, resulting in at least a model for each abducible that could not be defeated by contextual preference rules. In this case no active preferences hold and no observations are allowed, so the resulting stable models correspond to all the confirmed abducibles: `periapical_lesion`, `horizontal_fracture` and `vertical_fracture`. Since the system was not able to abduce just one model from a priori preferences, the choice mechanisms are activated and a new prospective search is launched for ways to defeat some of the models, as described in section 3.

However, the current knowledge state is insufficient to enact any kind of choice, so the system is forced to probe the external environment for additional information. This means that the oracle mechanisms are activated, and for the new observations the system decrees that experiments can now be performed. A new top-down query is launched, and as the derivation tree is traversed for new confirmation of relevant abducibles, opportunities to perform experiments are encountered. This is the case for the satisfaction of `expect_not(periapical_lesion)` which depends on `fracture_traces` and `not radiolucency`.

In an attempt to satisfy `fracture_traces`, ACORDA encounters an observable clause (cf. rule E). This clause depends on fracture traces being found on X-Ray imaging, so it probes the environment for just such an exam. No fracture traces are found, so periapical lesion can be confirmed under support from the oracle experiment. However, the subsequent experiment for `radiolucency` reveals positive results which can guarantee defeat of the other two abducibles, meaning that `periapical_lesion` is now the only expected and confirmed abducible. ACORDA can now successfully commit to a single diagnosis.

As mentioned above, in the cases where the system cannot guarantee a single abductive stable model after exhausting all the introspections, it queries the user to perform the final choice over the list of surviving abducibles. This behaviour could be refined, however, by launching separate simulations assuming each of the remaining solutions, and performing extra levels of prospective lookahead for additional ways to defeat further models. This addition is considered in the future work mentioned in section 5.

After the abduction of periapical lesion as the most likely cause for the sign of percussion pain on the patient's tooth, the attention of the system turns to satisfy a new observation for the source of such lesion. As such, a new cycle of introspection on top of these results can produce a more detailed diagnosis, as depicted below:

```
1. About to launch new introspection on selected active observables:
   [on_observable(prog,prog,periapical_lesion_source)]
2. Relevant abducibles for current introspection:
   [endodontic_lesion,periodontal_lesion]
3. Partial models remaining after a priori preferences:
   [[endodontic_lesion],[periodontal_lesion]]
4. About to launch new introspection on selected active observables:
   [on_observable(prog,prog,periapical_lesion_source)]
5. Confirm observation: pockets_check(gingival_pockets)
   (true, false or unknown)? true.
6. Confirm observation: periapical_xray(devitalization)
   (true, false or unknown)? true.
7. Relevant abducibles for current introspection:
   [endodontic_lesion,periodontal_lesion]
8. Partial models remaining after a priori preferences:
   [[periodontal_lesion]]
```

This time, the commitment to the abduction of `periapical_lesion` triggers the activation of program-to-program observable `periapical_lesion_source` (cf. rule C). The relevant expected abducibles that satisfy this observation are `endodontic_lesion` and `periodontal_lesion`, both being confirmed under the current knowledge state. Again, no active preferences hold a priori, that is, there are no preference rules relevant to the current abducibles which do not depend on any external observations. As a result, the Stable Models of the transformed partial residual program correspond once again to all relevant abducibles.

The choice mechanisms are activated and the top goal is relaunched in an attempt to acquire additional information, activating the oracles for external environment probing. The attempt to defeat `endodontic_lesion` calls for a gingival pockets measurement (cf. rule D), which reveals the existence of pockets in the vicinity of the patient's tooth. On the other hand, a periapical X-Ray is needed to confirm that no endodontic therapy was performed on that tooth. This experiment, however, reveals just that, and so ACORDA is unable to defeat the `endodontic_lesion` abducible using this *expect_not/1* clause.

Attempting to defeat `periodontal_lesion` by means of *expect_not/1* clauses proves impossible as well, due once again to the ambiguous results from the experiments. Both abducibles are again confirmed. However, a preference for `periodontal_lesion` is in place, given the existence of gingival pockets, so a unique Stable Model emerges, yielding the final diagnosis of `periodontal_lesion`.

The system could now be extended to handle different treatment hypotheses according to the result of the diagnosis. It would just be a matter of adding new triggers for additional observations that would represent the adequate treatment. Abducibles in this case would be the expected treatments and the preference model could be extended to include the patient's own preferences.

5 Conclusions and Future Work

The ACORDA project is still in its early beginnings, but even now it addresses several new pragmatic problems involving self-modifying logic programs by combining some of the best results from research and development of logic-based systems in the last decade. Preferential reasoning techniques are applied and the stable models semantics is used in an effective way by means of the XASP package. The relevance property enjoyed by top-down querying in XSB Prolog is crucial in preparation for our use of Smodels, and so we can expect to handle problems with a large rule-base, as we do not need to compute whole models, and none of those with irrelevant abducibles, in order to derive the relevant abductive stable models specific to the problem at hand.

This is one of the main problems which abduction over stable models has been facing, in that it always has to consider all the abducibles in a program and then progressively defeat all those that are irrelevant for the problem at hand. This is not so in the ACORDA system, since we begin by a top-down derivation that immediately constrains the set of abducibles that are relevant to the agent's observations and meta-goals. However, the encoding of abducibles as even-loops over default negation provides a declarative way in which to describe the abductive process, and the computation of stable models of the residual program is a natural way to obtain all the possible 2-valued models, considering also the a priori preferences, themselves coded as even loops over default negation.

Since virtually every element that is necessary to the process of abduction is encoded in the system's knowledge base, it automatically becomes a possible target of revision, meaning that the agent can update itself in order to change its preferences, and even the abductive rules, supervening old rules and adding new ones at will. Only the process of simulation in itself needs to remain separate, but that is just a practical consequence of the well-known self-reference problem. A program that simulates another program's future cannot be inside that very program. The process must be executed by an external system that is not included in the model that it is trying to simulate.

There are currently several avenues of improvement of the ACORDA system, orthogonal to several fields of active research in logic programming. To begin with, the stable models semantics may not be sufficient to model program evolution, since it does not guarantee the existence of models. Once we consider non-deterministic evolution of an agent, especially if it is expected to retrieve new rules from the environment or to compose old rules to form new ones or learn them, we cannot guarantee that odd-loops over default negation will not be part of the program somewhere in the future. If that happens, evolution is immediately halted, as stable models does not provide any model in that case.

It can be argued that inserting an odd-loop over default negation is simply a case of bad programming, and as such, it should never appear at all. However, if our agent makes a mistake, we want at least to give it an opportunity to revise that mistake, and not kill its evolution altogether. Fortunately, recent research on logic programming semantics has produced the Revised Stable Models [PP05] semantics, which elegantly solves this problem and brings a whole new host of desirable properties to the original SM semantics. Working implementations have already been developed and they will soon be integrated into the ACORDA system.

The abductive process and the system of a priori preferences can be improved as well, in order to allow for the abduction of a set of abducibles. It will be necessary to specify new ways in which we can express preferences over these sets, but work is already well underway in this regard. Abduction of multiple literals is, however, already possible in the current system, by making single abducibles themselves stand for sets of "abducible" literals. One just needs to carefully model all the relevant combinations of literals that one would be inclined to expect.

The integration of action prospections is also a must to tackle even more complex problems and applications, in order to deal with pre- and post-conditions. Namely, the pre-conditions of an action must be evaluated before the update for the action actually takes place, but the post-conditions must also be taken in consideration during the simulation and before the real action is executed. Also, we would like to extend the mechanism to an arbitrarily long sequence of actions, which would require an extensible amount of lookahead into the future.

In fact, there is a wide variety of problems for which we would like to be able to arbitrarily extend the future lookahead that is possible within the ACORDA system. One of the most important challenges in future developments is devising a way to do this that will also allow the specification of the degree of self-control over the amount of lookahead that the agent performs.

The need for a finer time stamping of events is also growing, since it would permit generalization of the validity of experiments. Instead of using tabling to hold the results of experiments, we would like to specify a time interval during which those results are assumed valid. This would be the first step in which the system could be extended to handle environment changes during simulation, and could even be used to model reactions of the system to such changes, or to introduce timeouts on the abductive process, if it proves to be taking too long.

Preferences over observables will also be desirable, since not every observation costs the same for the agent. Performing an X-Ray costs more than checking for tooth mobility or gingival pockets, and these differences should be modelled directly in the system. It would also be interesting to study more general ways of selecting the most interesting internal observations for ACORDA to pay attention to at each evolution step.

We have just started exploration of a territory that is vast in possibilities and, as such, we come out bearing more questions than answers. Even if we have a working system for such autonomous preferential reasoning and self-updating, the more interesting possibilities are still farther ahead, as we attempt to broach more and more of the goal of completely automated artificial reasoning, adumbrating its combinations.

6 Acknowledgments

We would like to thank Joana Nogueira for preparing and supplying scientific information for the case study on differential medical diagnosis, Pierangelo Dell'Acqua for constructive criticism on the definition of several specification and operational issues regarding the ACORDA system, and a number of colleagues at DEIS, U. Bologna, for prior stimulating discussions.

References

[ABLP02] J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Procs. of the 8th European Conf. on Logics in Artificial Intelligence (JELIA'02)*, LNCS 2424, pages 50–61, Cosenza, Italy, September 2002. Springer.

[ALP$^+$00] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *J. Logic Programming*, 45(1-3):43–70, September/October 2000.

[AP00] J. J. Alferes and L. M. Pereira. Updates plus preferences. In M. O. Aciego, I. P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Procs. of JELIA'00*, LNAI 1919, pages 345–360, Málaga, Spain, 2000. Springer.

[APS04] J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, July 2004.

[CSW] L. Castro, T. Swift, and D. S. Warren. *XASP: Answer Set Programming with XSB and Smodels.* `http://xsb.sourceforge.net/packages/xasp.pdf`.

[DMS] N. Drakos, R. Moore, and H. Singh. *The XSB System Programmer's Manual.* `http://xsb.sourceforge.net/manual1/`.

[DP05] P. Dell'Acqua and L. M. Pereira. Preferential theory revision. In L. M. Pereira and G. Wheeler, editors, *Procs. Computational Models of Scientific Reasoning and Applications*, pages 69–84, Universidade Nova de Lisboa, Lisbon, Portugal, September 2005.

[DP07] P. Dell'Acqua and L. M. Pereira. Preferential theory revision (extended version). *J. Applied Logic*, 2007. Forthcoming.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th Intl. Logic Programming Conf.*, pages 1070–1080. MIT Press, 1988.

[NS97] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *4th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning*, LNAI 1265, pages 420–429, Berlin, 1997. Springer.

[PP05] L. M. Pereira and A. M. Pinto. Revised stable models - a semantics for logic programs. In *12th Portuguese Intl. Conf. on Artificial Intelligence (EPIA'05)*, LNAI 3808, pages 29–42, Covilhã, December 2005. Springer.

[Swi99] T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240, 1999.

[vGRS91] A. van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.