# Personal Perspective on the Development of Logic Programming Based KR Languages

Michael Gelfond
Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
michael.gelfond@ttu.edu

May 9, 2011

## 1 Introduction

This paper is a short comment on some of my work on the development of knowledge representation (KR) languages. The story it presents can be viewed as a mixture of real history and rational reconstruction of this work. Since my goal is simply to articulate the methodological problems I encountered in this research and my attempts at their solutions, the paper almost entirely ignores contributions made by other researchers. The hope, of course, is that sharing even an incomplete story can shed some light on the current state of the field and may even be helpful to other people confronted with similar problems.

I first became interested in the problem of knowledge representation in the early eighties when Vladimir Lifschitz came back to El Paso from Stanford Summer School with fascinating stories about defaults, non-monotonic logics and circumscription he learned in John McCarthy's course on AI. I attended the same school the following year and took the same course together with a course on Logic Programming taught by Kenneth Bowen. In that class I wrote my first Prolog program and got introduced to Prolog's treatment of negation. Even though in the beginning I felt that all the problems of non-monotonic reasoning could be solved in the framework of classical first-order logic, I soon realized that we may be dealing with a new phenomenon requiring different mathematical models and techniques. Since that time I have been looking for ways of formalizing various aspects of non-mathematical (especially common-sense) knowledge and reasoning.

My first encounter with methodological problems in this research occurred when I read the conference version of [24]. In this paper Hanks and McDermott introduced a simple temporal projection problem, which later became known as the Yale Shooting Problem. They considered several possible formalizations of

this problem in known non-monotonic formalisms and showed that these formalizations did not produce the expected results. Based on this the authors concluded that temporal projection could not be naturally represented in these formalisms. I was surprised by the conclusion since it was not clear to me if the *problems were caused by the formalisms or by the particular methods of representing knowledge chosen by the authors*. The second possibility seemed to be supported by the simple and natural representations of the Yale Shooting problem in Reiter's Default Logic [38], in Moore's Autoepistemic Logic [31], and in Logic Programming, which were found almost immediately after the publication of the paper (see [32], [19] and [14]). This alerted me early to the danger of premature conclusions. In particular it became clear that the *syntax and semantics of non-monotonic formalisms should be developed in conjunction with the development of the corresponding knowledge representation methodology.*

The need to better understand the phenomenon of non-monotonicity has not been limited to the area of knowledge representation. In the eighties logic programing language Prolog, developed by Colmerauer, Roussel, Kowalski, Warren and others, became a serious and powerful programming tool used for multiple practical applications. The language was especially attractive since it had a strong declarative component. A program in the original, "pure" Prolog could be seen as a collection of definite clauses of first-order logic with the minimal Herbrand model of the program providing its semantics. This minimal model could be viewed as the *intended model* of the program. From the beginning, however, Prolog had important non-logical features, including new non-monotonic logical connective "*not*" often referred to as *negation as failure* or *default negation*. Its original semantics was defined procedurally — *not* $p$ was understood as finite failure to prove $p$ by a particular proof algorithm called $SLDNF$ resolution. To maintain the declarative character of Prolog, it was important to find declarative semantics of this connective. By the mid eighties there was a substantial amount of work in this direction. One approach, suggested by K. Clark [12], viewed a logic program $\Pi$ as a shorthand for a first-order theory called Clark's completion of $\Pi$ with models of the completion defining the semantics of $\Pi$. Another approach aimed at expanding the definition of intended model from pure Prolog to programs with default negation. An influential paper by Apt, Blair and Walker [1] succeeded in defining such an intended model for so called stratified logic programs.

Perhaps somewhat surprisingly work on non-monotonic logic was developing more or less independently from that on the semantics of default negation in logic programming. However, there was a growing conviction among a small group of researchers in both areas that there were deep connections between them.

In 1987 I made some progress in discovering one such connection. A simple mapping of logic programs into Autoepistemic Logic [18] allowed the use of logic programming to answer queries to a non-trivial class of autoepistemic theories and to expand the intended model semantics of stratified programs. The

resulting semantics became known as Stable Model semantics of logic programs [20].

At about the same time, Well-Founded Semantics of logic programs was introduced by Van Gelder, Ross, and Schlipf [17]. Both semantics coincided for stratified programs but, in more general cases, there were substantial differences between them. *Immediately after this introduction, discussion began about the relative merits of both semantics from the KR stand point. Differences of opinion were caused not only by personal intuitions and tastes, but also by different views on the goals and proper methodology for KR research and on the proper role of KR languages.* These differences, however, were often hidden in the background and very rarely articulated (at least outside of the anonymous reviewing process). This paper is an attempt to partially fill this gap. We do not necessarily need to reach consensus on the subject — in fact I believe that such a consensus may be harmful. But I also believe that asking and answering these questions is essential for every researcher, and that seeing how they were answered by others can be helpful.

## 2 Stable versus Well-Founded Semantics

In this section I'll recall the discussions on comparative merits of stable and well-founded semantics from the standpoint of knowledge representation which started around 1988. (Of course a large part of what was happening in research on semantics of logics programs at the time, including all other interesting semantics, will be omitted. It will be great to see other people's recollections of this.) I'll start with (a possibly incomplete) *list of different criteria which were used to evaluate KR languages*:

1. Clarity: logical connectives of a language should have a reasonably clear intuitive meaning.
2. Elegance: the corresponding mathematics should be simple and elegant.
3. Expressiveness: a KR language should suggest systematic and elaboration-tolerant representations of a broad class of phenomena of natural language. This includes basic categories such as belief, knowledge, defaults, causality, etc.
4. Relevance: a large number of interesting computational problems should be reducible to reasoning about theories formulated in this language.
5. Efficiency: Reasoning in the language should be efficient.
6. Regularity of entailment: entailment relation $\models$ of the language should satisfy some natural properties, e.g. cautious monotonicity: if $T \models F$ and $T \models G$ then $T \cup \{F\} \models G$.
7. Consistency: every program written in the language should be, in some sense, consistent.

8. Supra-classicality: the new language should extend first-order classical logic.

To decide which of the above criteria were important and which were less so, I needed to have a clear understanding of why I wanted to represent knowledge. Even though the details of my views on the subject evolved substantially, the basic answer seems to remain unchanged. I had two closely interrelated but distinct goals. *As a logician I wanted to better understand the basic commonsense notions we use to think about the world: beliefs, knowledge, defaults, causality, intentions, probability, etc., and to learn how one ought to reason about them. As a computer scientist I wanted to understand how to build software components of agents — entities which observe and act upon an environment and direct its activity towards achieving goals* . It is worth noticing that at the time I viewed even the simplest programs as agents. A program usually gets information from the outside world, performs an appointed reasoning task, and acts on the outside world by printing the output, making the next chess move, or starting a car. If the reasoning tasks it performs are complex and lead to nontrivial behavior, we call the program intelligent. If the program nicely adapts its behavior to changes in its environment, it is called adaptive, etc. It is possible that I developed this view because in my first real programming work I dealt with a large control system for paper production, and my first programming assignment was to figure out why some bulb didn't light when it should, and to make it function properly. Clearly my program was supposed to observe, think, and act on the environment to achieve certain goal. To my surprise I later learned that for many people this view of programs seemed unnatural.

Applied to the area of logic programming and deductive databases, this view lead to the notion of *agent* which maintains its knowledge base (KB), represented as a logic program, and is capable of expanding it by new information and of answering questions about its knowledge. The Closed World Assumption [37] built into the semantics of logic programs made the agent's answers defeasible. An agent with a simple KB = $\{p(a)\}$ will answer *no* to a query $?p(b)$[1]. If in communicating with the outside world the agent will learn that $p(b)$ is true, it will be able to nicely incorporate this information into its knowledge base. The new KB will consist of $\{p(a), p(b)\}$. Now the query $?p(b)$ will be answered with *yes*. This is a *typical example of non-monotonicity which prompted me to interpret the agent's conclusions as statements about its beliefs (as opposed to statements about actual truth or falsity of propositions).* Other people had different views. The fact that none of these views were normally clearly articulated in print caused a substantial amount of misunderstanding. *My general assumption about the nature of agent and its knowledge base influenced the definition of stable model.* Let us recall that, informally, stable models are collections of ground atoms which:

• Satisfy the rules of Π.

[1]I assume here that $b$ belongs to the signature of the program. Similar assumptions will be made throughout the paper.

• Adhere to the *rationality principle* which says: *Believe nothing you are not forced to believe.*

The mathematical definition of stable models captures this intuition.

Now it was *time to evaluate semantics of logic programs with respect to the above criteria.* I was satisfied with the intuitive *clarity* and mathematical *elegance* of the stable model semantics. In this respect I had more difficulties with the well-founded semantics. For my taste the original definition of well-founded semantics looked too complex and not as declarative as I wanted. Later it was demonstrated that well-founded semantics can be viewed as a three-valued version of stable model semantics, but I still had difficulties with the declarative meaning of the third value. I think that reasonable people can disagree with this evaluation, but it was clear that in both semantics we only started to address the criterion of *expressiveness* of our languages. We better understood rational beliefs and reasoning with defaults, but even this substantial advance was hampered by *inability of the language to adequately represent incomplete information.* Existence of multiple stable models allowed some representation of incompleteness, but it seemed ad hoc. Moreover, the Closed World Assumption was a built-in feature of the stable model semantics. From my standpoint it was the major drawback of the language. Despite its third value the well-founded semantics didn't seem to fare any better. Not much was available to us at the time to satisfy the fourth criterion —*relevance* of our languages for computing. We were able to use datalog and logic programming algorithms to build agents capable of answering sophisticated queries but that was it. The situation changed dramatically with the development of powerful answer set solvers such as Smodels and DLV [35],[33] and with mathematical results reducing planning [39], diagnostics [4], and many other non-trivial problems to computing (parts of) answer sets of logic programs. In 1989 all I could do was to give the stable model semantics an "incomplete" on the fourth criterion. The fifth requirement — that of the *efficiency* of reasoning — was interpreted differently by different people. One group strongly believed that stable model semantics didn't satisfy this criterion since even for programs with finite Herbrand models the problem of finding a stable model of a program is NP-complete. (Computing the well-founded model is quadratic.) Another view understood complexity in a less stringent way. Even though Pure Prolog is undecidable in the presence of function symbols, many practical programs were successfully written and run with the standard Prolog interpreter (which, under reasonable conditions, is sound with respect to both semantics). For me the second view was obviously correct. (I have to confess that I still do not really understand the arguments for the first one.) Requirement six – the *regularity of entailment* – has been much more appealing. Unfortunately, as was shown in [17], stable model semantics does not satisfy even the simple property of cautious monotonicity. This was unfortunate but not very surprising. When I first studied calculus, my intuition revolted against continuous functions that are nowhere differentiable, but I was glad that existence of such counter-intuitive examples didn't force the mathematicians to change the notion of continuous function. Still the fact that the entailment

relation under the well-founded semantics is cautiously monotonic was rather appealing. Well-founded semantics also satisfies criterion of *consistency*. Every logic program has the well-founded model[2]. Program $p \leftarrow not\ p$ has no stable models. This fact, however, didn't seem unnatural to me. I thought that the rule which says something like "if there is no reason to include $p$ in your set of beliefs then include it in this set" can be viewed by a rational reasoner as non-sensical. The last criterion, *supra-classicality*, was not satisfied by the language of logic programs under any type of semantics. Overall *it was clear that making a definite choice between the two semantics was premature. We needed to expand the original language of logic programs to allow better expressiveness.* In [21] Vladimir Lifschitz and I came up with the language of extended logic programs which is now known as A-Prolog, ANS-Prolog, or Answer Set Prolog.

# 3 Answer Set Prolog (ASP)

In order to better understand the new language, let us recall that the stable model semantics was defined for programs consisting of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n. \tag{1}$$

where $a$'s are atoms of the program's signature. Rules with variables are viewed as shorthands for the set of their ground instantiations, so we assume that our atoms and rules are ground.

**Definition 1** *[Stable Models]*
An atom $a$ is true in a set of ground atoms $S$ if $a \in S$. Otherwise, $a$ is false in $S$. An expression $not\ a$ is true in $S$ if $a \notin S$. Otherwise, $not\ a$ is false in $S$. A set $S$ satisfies a rule (1) if $a_0$ is true in $S$ whenever all the statements in the body of the rule are true in $S$. For programs without default negation *not*, the stable model of the program is defined as the minimal set of atoms satisfying the program's rules. A set $S$ of atoms is a stable model of an arbitrary program $\Pi$ if it is a stable model of the reduct $\Pi^S$ of $\Pi$ with respect to $S$, which is obtained from $\Pi$ by

1. removing all the rules whose body contain $not\ a$ such that $a \in S$;
2. removing the remaining occurrences of expressions of the form $not\ a$.

A program $\Pi$ entails $p$ if $p$ is true in every stable model of $\Pi$; $\Pi$ entails $\neg p$ if $p$ is false in every stable model of $\Pi$. In the former case, $\Pi$ answer *yes* to query $?p$. In the latter, the answer to this query is *no*. Otherwise, the answer is *unknown*.

Rules of Answer Set Prolog are substantially more general. They have the form:

$$l_0 \text{ or } \ldots \text{ or } l_i \leftarrow l_{i+1}, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n. \tag{2}$$

[2]This property, of course, disappears if the language allows rules with the empty head. This is not true for the stable model semantics.

where $l$'s are literals of the program's signature. (Note that the head of the rule can be empty.) A literal is defined as an atom $a$ or its classical negation $\neg a$. (A literal possibly preceded by default negation is often referred to as an extended literal.) Statement $\neg a$ says that $a$ is false, while *not* $a$ only states that there is no rational support for believing in $a$. Symbol *or* denotes a new logical connective, called *epistemic disjunction*. Statement $a$ *or* $b$ indicates that the reasoner must believe $a$ or must believe $b$. (Hence "$a$ *or* $\neg a$" is not a tautology.) The new language generalizes classical Prolog programs with default negation as well as disjunctive logic programs in the style of [30] which consist of rules of the form

$$a_0 \text{ or } \ldots \text{ or } a_i \leftarrow a_{i+1}, \ldots, a_m. \tag{3}$$

As before the rules of the program are viewed as constraints on the sets of beliefs which can be formed by a reasoner who adheres to the rationality principle. But this time beliefs are represented by consistent sets of ground literals called *answer sets* of the program. The *definition of answer set requires a minimal change in the definition of stable model*. All we need is a natural definition of truth and the consistency requirement.

**Definition 2** *[Answer Set]*
A literal $l$ is true in a set of literals $S$ if $l \in S$; *not* $l$ is true in $S$ if $l \notin S$; $l_1$ *or* $l_2$ is true in $S$ if either $l_1$ or $l_2$ is true in $S$. An answer set of a program without default negation *not* is a minimal consistent set of literals satisfying the rules of $\Pi$. For an arbitrary program it is an answer set of the reduct $\Pi^S$ defined as in Definition 1.

The definition of entailment and that of answer to a query remain unchanged.

*Despite this similarity in the definitions, introduction of new logical connectives and the shift from sets of atoms to sets of literals had a substantial impact on the language*. Consider for instance a simplest possible

$$\Pi_1 = \{p(a)\}$$

The answer set of this program is $S = \{p(a)\}$. So is its stable model. In the latter case $p(b)$ is false in $S$ and, hence, the agent's answer to query $?p(b)$ will be *no*. In the former case, however, $p(b)$ is neither true nor false in $S$ and, hence, the answer to $p(b)$ under the answer set semantics will be *unknown*. In other words the *Closed World Assumption is no longer part of the semantics of the new language*. It is important to see that in Answer Set Prolog this assumption for a relation $p$ can be expressed by a statement:

$$\neg p(X) \leftarrow not\ p(X).$$

The statement says that $p(x)$ which is not believed to be true should be false. Program $\Pi_2$ obtained by adding this rule to $\Pi_1$ will answer *no* to a query $?p(b)$. Now the agent associated with the program believes that $p(b)$ is false. Of course, if $p(b)$ is learned to be true, the agent will simply add it to its knowledge base.

The previous conclusion will be withdrawn. This simple example shows that the agent using a knowledge base written in ASP can gracefully accommodate new information and change its beliefs.

After completion of the language design part of our work, we needed to clearly understand what should be done to evaluate our language. It became clear rather early that ASP preserves the degree of clarity of intuition and mathematical elegance of the original language. But much time and effort was required to show that:

1. ASP increased our ability to express basic commonsense notions we use to think about the world.
2. A large number of interesting computational problems can be reduced to reasoning in ASP.
3. Efficient ASP reasoning algorithms can be found and built into the systems capable of solving these problems.

*To test the expressibility of the new language, we first concentrated on representing defaults* — statements of the form "*Normally elements of class c satisfy property p.*" The importance of defaults for knowledge representation and AI has been known for a long time. A large part of our education seems to consist of learning various defaults, their exceptions, and the skill of reasoning with them. Since defaults did not occur in the language of mathematics, they were not studied in classical mathematical logic. However, they play a very important role in everyday commonsense reasoning, and present a considerable challenge to AI researchers. Even though in the eighties remarkable progress had been made addressing this challenge, I thought that more work was needed to find a fully satisfactory solution. The first question was how to represent a default. In [6] we decided on the following general representation:

$$\begin{array}{lll} p(X) & \leftarrow & c(X), \\ & & not\ ab(d(X)), \\ & & not\ \neg p(X). \end{array}$$

where $d$ is the default's name. The representation uses the abnormality predicate used in work by J. McCarthy [28] for representing exceptions to defaults, and the default representation used in Reiter's Default Logic [38]. (Since ASP programs without disjunction can be viewed as theories of Reiter's Default Logic, such a representation was rather natural.) We also considered two types of exceptions to defaults. A subclass $c_1$ of $c$ is called a *strong exception* to default $d(X)$ if elements of $c_1$ do not satisfy property $p$; $c_1$ is called a *weak exception* if the default shall not be applied to its elements. Strong exceptions can be represented by the rules:

$$\neg p(X) \leftarrow c_1(X).$$

$$ab(d(X)) \leftarrow not\ \neg c_1(X).$$

In our representation $ab(d(X))$ holds if the default $d$ shall not be applied to $X$. The second rule says that if $x$ may belong to $c_1$, then application of default $d$ to $X$ should be stopped. Of course if the information about membership in class $c_1$ is complete, the second rule is unnecessary. The weak exceptions to $d$ are represented by the rule:

$$ab(d(X)) \leftarrow not\ \neg c_1(X).$$

If information about membership in class $c_1$ is complete, then $not\ \neg c_1(X)$ in the body of the rule can be replaced by $c_1(X)$. From the mapping of ASP programs without disjunction into Reiter's default theories, we know that all of the above rules can be expressed in Reiter's default logic. The situation changes when we decide to consider theories containing disjunctions. Encoding of weak exceptions in ASP can also be done using the rule:

$$p(X) \text{ or } \neg p(X) \leftarrow not\ \neg c_1(X).$$

which, in some situations, may be preferable to the one using $ab$; however, it becomes less natural in Reiter's logic. In addition, representing alternatives using classical disjunction in default logic does not allow for default reasoning by cases. Consider, for instance, a program

$$\begin{array}{lcl} p_1(X) & \leftarrow & c_1(X), \\ & & not\ \neg p_1(X). \\ p_2(X) & \leftarrow & c_2(X), \\ & & not\ \neg p_2(X). \\ q(X) & \leftarrow & p_1(X). \\ q(X) & \leftarrow & p_2(X). \end{array}$$

used together with a disjunction

$$c_1(a) \text{ or } c_2(a).$$

Clearly the program entails $q(a)$ which corresponds to our intuition about proper reasoning with defaults. If, however, the above disjunction was understood as a classical statement in the first-order logic part of the corresponding default theory, the conclusion would not be reachable.

The above method of encoding exceptions to defaults could also be used to specify preferences between conflicting defaults. Assuming that preferences and conflicts are fully specified this can be done, say, by the rule:

$$\begin{array}{lcl} ab(d_1(X)) & \leftarrow & prefer(d_1(X), d_2(X)), \\ & & in\_conflict(d_1(X), d_2(X)). \end{array}$$

In [23] we tested our representation of defaults and their exceptions using SLG [11] — one of the first reasoning systems capable of dealing with logic programs with multiple stable models. The system was still a prototype and not very easy to use, but the ability to implement such reasoning was in itself exciting.

Of course more-powerful systems, like Smodels and DLV, followed quickly and writing programs capable of sophisticated default reasoning became almost a routine task.

Unfortunately, our representation of defaults in ASP had its difficulties. *While we were able to represent weak and strong exceptions to defaults, we still were not able to deal with situations when contradictions were found not with the conclusion of the default, but with the consequences of this conclusion.* We refer to such exceptions as *indirect*. To see the problem let us consider program $I_1$ consisting of rules

$$\begin{array}{lcl} p(X) & \leftarrow & c(X), \\ & & not\ ab(d(X)), \\ & & not\ \neg p(X). \\ q(X) & \leftarrow & p(X). \\ c(a). & & \end{array}$$

Clearly, $I_1$ entails $q(a)$. But this program can not accept an update of the form $\neg q(a)$. The attempt to add this fact to the program leads to inconsistency. This problem was one of the reasons for the development of an extension of ASP called CR-Prolog [5, 2], which we will discuss in the next section.

In [22] we also looked at the possibility of *using ASP for representing causal effects of actions*. This started important work on action languages and lead to the establishment of a close relationship between reasoning about actions and ASP, which showed that ASP is capable of elegantly expressing direct and indirect effects of actions, and addressing the frame and ramification problems. For instance, the sentence

$$a \text{ causes } f$$

of action language $\mathcal{B}$ [27] says that if $a$ were executed in some state, then fluent $f$ would be true in the system's successor states. This causal law can be written as the ASP rule

$$holds(f, I+1) \leftarrow occurs(a, I).$$

where $I$ ranges over steps of the system's trajectory. Of course if $a$ were to make $f$ false, we would simply replace the head of the rule by $\neg holds(f, I+1)$. Another typical causal law of $\mathcal{B}$ expresses the relationship between fluents. It has a form

$$l_0 \text{ if } l_1, \ldots, l_n$$

and says that any state of the system in which fluent literals $l_1, \ldots, l_n$ are true should also satisfy fluent literal $l_0$. This law can be naturally represented by the ASP rule

$$h(l_0, I) \leftarrow h(l_1, I), \ldots, h(l_n, I).$$

where for every fluent $f$, $h(f, I)$ denotes $holds(f, I)$ and $h(\neg f, I)$ denotes $\neg holds(f, I)$. The classical frame problem can be solved in ASP by the Inertia Axiom:

$$holds(F, I+1) \leftarrow holds(F, I), not\ \neg holds(F, I+1).$$

$$\neg holds(F, I+1) \leftarrow \neg holds(F, I), not\ holds(F, I+1).$$

which formalizes the default "*Things tend to stay as they are.*" The formalization of dynamic domains outlined above successfully solved the frame and ramification problems which caused substantial difficulties in many previous attempts to formalize reasoning about actions and change. The success was mainly due to the ability of ASP to represent defaults and the use of non-contrapositive[3] rules which seem to better capture causal relations than classical implication. *This work clarified the nature of reasoning about dynamic domains and allowed to reduce classical AI problems such as planning and reasoning to computing answer sets of logic programs.*

Of course the two examples above do not cover all the basic concepts we would like to be able to represent in ASP. Later, substantial progress was made in using ASP and its extensions to reason about knowledge, intentions, probabilities, etc. But at the time *it was important to go beyond the first step of our methodology and actually check if the language could be made useful for the design of intelligent agents.* Our initial work concentrated on tasks which could be reduced to answering queries to a knowledge base. A typical example of such work included a paper [40] in which ASP was used to *expand deductive databases with the ability to reason about various forms of null values.* I believe that, conceptually, we succeeded in showing that this can be done in a principled way, but the absence of access to programs with efficient interfaces between deductive and relational databases precluded us from trying to use these ideas to build practical prototypes. (Part of the difficulty was probably related to the fact that some work on deductive databases was done by companies. As a result we often heard rumors about such systems but I am still not sure if they really existed at the time. Recently I was happy to learn that the DLV group implemented and used an efficient interface with standard relational databases [15]).

Our second attempt to investigate applicability of ASP to the design of intelligent systems was more successful. It started in the mid nineties in cooperation with United Space Alliance (USA) — the company responsible at the time for day to day operations of the Space Shuttle. The USA people came to us with the following problem. The Shuttle has a reactive control system (RCS) that has primary responsibility for maneuvering the aircraft while it is in space. It consists of fuel and oxidizer tanks, valves and other plumbing needed to provide propellant to the maneuvering jets of the Shuttle. It also includes electronic circuitry: both to control the valves in the fuel lines and to prepare the jets to receive ring commands. Overall, the system is rather complex, in that it includes 12 tanks, 44 jets, 66 valves, 33 switches, and around 160 computer commands (computer-generated signals). When an orbital maneuver is required, the astronauts must configure the RCS accordingly. This involves changing the position of several switches, which are used to open or close valves or to energize the

[3] To see that ASP rules are non-contrapositive it is enough to note that programs $\Pi_1 = \{p \leftarrow q.\ \ \neg p\}$ and $\Pi_2 = \{\neg q \leftarrow \neg p.\ \ \neg p\}$ are not equivalent, i.e. have different answer sets.

proper circuitry. Normally, the sequences of actions required to configure the RCS are pre-determined before the beginning of the mission, and the astronauts simply have to search for the sequence in a manual. However, faults (e.g. the inability to move a switch) may make these pre-scripted sequences of actions inapplicable. The number of possible sets of failures is too large to plan in advance for all of them. In this case, the astronauts communicate the problem to the ground flight controllers, who come up with a new sequence of actions to perform the desired task. The USA wanted to *develop software to automatically check if the plans found by controllers are correct and do not cause dangerous side-effects.* Developing this software required methodology for modeling of and reasoning about complex dynamic systems. To do the checking the system had to have knowledge of the initial situation, the causal laws that rule the evolution of the domain and the operational knowledge of the controllers. We were delighted to discover that Answer Set Prolog proved to be suitable for representing this knowledge. Moreover, the representation was written in the form of an acyclic logic program which allowed us to check correctness of plans by running a standard Prolog interpreter [41]. Mathematical results establishing the relationship between descriptions of dynamic domains in high-level action languages and their logic programming counterparts, together with the work done in the Prolog community on termination and soundness of the interpreter, allowed us to significantly reduce the risk of programming errors. Overall our first experiment was successful — the resulting system was comparatively small, efficient, elaboration tolerant, and understandable to the USA people. However, our attempts to expand the system's functionality by teaching it to automatically search for the plans using a Prolog interpreter failed. In the original system plans were represented by terms and hence, in principle, could be found by the resolution mechanism. We were, however, *not able to overcome technical difficulties (including those related to floundering) and, hence, could not use our experiment to conclude that Answer Set Prolog can indeed be used as a practical tool for building multi-purpose intelligent agents.*

Fortunately the situation changed very substantially with the new breakthroughs in the areas. The late nineties witnessed the appearance of new powerful algorithms and systems for computing stable models/answer sets of logic programs with finite Herbrand universes. This was accompanied by the change in emphasis from Prolog-style methodology of query-answering to the new paradigm [26, 34] of reducing computational problems to finding answer sets of a program and computing these sets using answer set solvers. For me this development was a very pleasant surprise. I expected the development of really efficient systems of this sort to take much more time. However, after talking to Victor Marek and Mirek Truszczynski at the Kentucky workshop in 1997, I changed my mind. Together with my students, we decided to try the new technique on our Space Shuttle project. Reduction of planning to computing answer sets was already discussed in earlier papers and proving correctness of this approach in our context was not too difficult. The method worked. Reasonable plans were normally found in a matter of seconds. (From the USA standpoint, ten minutes was a

good time.) We substantially expanded the system by adding more knowledge about RCS as well as the ability to find diagnosis for unexpected observations. From then on we used the system, called the USA advisor, to test our new reduction and reasoning algorithms and new KR languages.

From my standpoint the *work on USA advisor ended the first stage of our research program of evaluating the quality of ASP as a knowledge representation language*. The work went through several stages which can be formulated as follows:

1. Development of the syntax and semantics of the language accompanied by the investigation of methodology of the language used for knowledge representation.

   Here the emphasis is on the ability to model the basic conceptual notions, faithfulness to the intuition, and mathematical accuracy and elegance.

2. Evaluation of the language and its KR methodology by its use in designing small experimental systems capable of performing intelligent tasks.

   The emphasis here is on the ability of the language to guide our design, generalizability of solutions, and discovery of new phenomena (or a new perspective on an old one).

3. Evaluation of the language by its use in design and implementation of mid-size practical intelligent software systems.

   Emphasis here is on efficiency, correctness, and degree of elaboration tolerance.

Of course none of these steps was possible without the development of a mathematical theory of the language. The last two steps were also impossible without development of reasoning systems for the language. We were fortunate that such systems were made available to us thanks to the first class work done by other researchers.

# 4 Expanding ASP by abduction: CR-Prolog

Now it was time to concentrate on the KR problems which were not solved by ASP. One such problem, already discussed above, was the inability of ASP to represent indirect exceptions to defaults. This observation led to the development of a simple but powerful extension of ASP called CR-Prolog (or ASP with consistency-restoring rules). To illustrate the basic idea, let us go back to program $I_1$ from Section 2. We have already seen that this program, extended by an observation $\neg q(a)$, is inconsistent. There, however, seems to exists a commonsense argument which may allow a reasoner to avoid inconsistency, and to conclude that $a$ is an indirect exception to the default. The argument is based on the **Contingency Axiom** for default $d(X)$ which says that "*Any element of class c can be an exception to the default* $d(X)$*, which says that elements of class*

*c normally have property p, but such a possibility is very rare and, whenever possible, should be ignored.*" One may informally argue that since the application of the default to $a$ leads to a contradiction, the possibility of $a$ being an exception to $d(a)$ cannot be ignored and, hence, $a$ must satisfy this rare property.

To allow formalization of this type of reasoning, we expand the syntax of ASP by rules of the form

$$l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_k, not\ l_{k+1}, \ldots, not\ l_n. \tag{4}$$

where $l$'s are literals. Intuitively, rule (4), referred to as *consistency restoring* rule (cr-rule), says that if the reasoner associated with the program believes the body of the rule, then it "may possibly" believe its head[4]. However, this possibility may be used only if there is no way to obtain a consistent set of beliefs by using only regular rules of the program. The semantics of CR-Prolog is given with respect to a partial order, $\leq$, defined on sets of cr-rules. This partial order is often referred to as a **preference relation**. We will need the following notation.

The set of regular rules of a CR-Prolog program $\Pi$ is denoted by $\Pi^r$; by $\alpha(r)$ we denote a regular rule obtained from a consistency restoring rule $r$ by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; $\alpha$ is expanded in a standard way to a set $R$ of cr-rules, i.e. $\alpha(R) = \{\alpha(r) \ : \ r \in R\}$. As in the case of ASP, the semantics of CR-Prolog is defined for ground programs. A rule with variables is viewed as shorthand for a schema of ground rules.

**Definition 3** *[Answer Sets of CR-Prolog]*
A minimal (with respect to the preference relation of the program) collection $R$ of cr-rules of $\Pi$ such that $\Pi^r \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an *abductive support* of $\Pi$.
A set $A$ is called an *answer set* of $\Pi$ if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support $R$ of $\Pi$.

Consider, for instance, the following CR-Prolog program:

$p(a) \leftarrow not\ q(a).$
$\neg p(a).$
$q(a) \stackrel{+}{\leftarrow}.$

It is easy to see that the regular part of this program (consisting of the program's first two rules) is inconsistent. The third rule, however, provides an abductive support which allows to resolve inconsistency. Hence the program has one answer set $\{q(a), \neg p(a)\}$.

Now let us show how CR-Prolog can be used to represent defaults and their indirect exceptions. The CR-Prolog representation of default $d(X)$ may look as

[4]It may be worth noting that intuitively, the rule $l_0$ or $l_1 \stackrel{+}{\leftarrow} body$ can be replaced by two rules $l_0 \stackrel{+}{\leftarrow} body$ and $l_1 \stackrel{+}{\leftarrow} body$. Hence we do not allow disjunction in the heads of cr-rules.

follows

$$\begin{array}{rcl} p(X) & \leftarrow & c(X), \\ & & not\ ab(d(X)), \\ & & not\ \neg p(X). \\ \neg p(X) & \stackrel{+}{\leftarrow} & c(X). \end{array}$$

The first rule is the standard ASP representation of the default, while the second rule expresses the Contingency Axiom for default $d(X)$[5]. Consider now a program obtained by combining these two rules with an atom

$$c(a).$$

Assuming that $a$ is the only constant in the signature of this program, the program's answer set will be $\{c(a), p(a)\}$. Of course this is also the answer set of the regular part of our program. (Since the regular part is consistent, the Contingency Axiom is ignored.) Let us now expand this program by the rules

$$\begin{array}{rcl} q(X) & \leftarrow & p(X). \\ \neg q(a). & & \end{array}$$

The regular part of the new program is inconsistent. To avoid the problem we need to use the Contingency Axiom for $d(a)$ to form the abductive support of the program. As a result the new program has the answer set $\{\neg q(a), c(a), \neg p(a)\}$. The new information does not produce inconsistency as in the analogous case of ASP representation. Instead the program withdraws its previous conclusion and recognizes $a$ as a (strong) exception to default $d(a)$.

The above examples had only one possible resolution of the conflict and, hence, its abductive support did not depend on the preference relation of the program. When this is not the case, preferred abductive supports are used to form the program's answer sets.

The ability to encode rare events which could serve as possible exceptions to defaults proved to be very useful for various knowledge representation tasks. For instance, in reasoning about actions we assume that normally the agent is aware of all the relevant actions occurring in the domain. This assumption can be expressed as

$$\neg occurs(A, I) \leftarrow not\ occurs(A, I).$$

Here $A$ is a variable for actions and $I$ ranges over the steps of the agent's trajectory. The consistency restoring rule

$$occurs(A, I) \stackrel{+}{\leftarrow} agent\_action(A), I > n.$$

where $n$ is the last step of the current trajectory says that any agent action may occur in the future. Used together with the usual planning constraint which

[5]In this form of Contingency Axiom, we treat $X$ as a strong exception to the default. Sometimes it may be useful to also allow weak indirect exceptions; this can be achieved by adding the rule: $ab(d(X)) \stackrel{+}{\leftarrow} c(X)$.

states that failing to achieve the goal is not an option, this rule can be used to find optimal plans. Similar (but more sophisticated) techniques were used in [3] to improve quality of plans found by the USA advisor. A similar rule

$$occurs(A, I) \stackrel{+}{\leftarrow} exogenous\_action(A), I < n.$$

where by exogenous action we mean actions performed by nature or other agents in the domain, can be used to do diagnostics. In this case the discrepancy between the predicted value of the fluent and its observed value will be explained by some missed occurrences of unobserved actions which restore the program's consistency. Despite the fact that CR-Prolog proved to be a sufficiently simple and useful tool for knowledge representation which nicely combine traditional ASP reasoning with some form of abduction, the CR-Prolog-related research program is far from complete. Even though there is a meta-level implementation of CR-Prolog which was sufficiently efficient to be used for some practical applications, it is still too slow for others. The proper implementation should be tightly coupled with efficient ASP solvers — something we currently cannot do at TTU. A more important and substantially more difficult problem is related to finding proper ways of defining preference relation $\leq$. Should preference be defined between pairs of cr-rules and expanded to the sets of such rules? Should the preference be dynamic (i.e. allowed to be defined by the program's rules) or static? Should we have the preference built into the semantics of CR-Prolog? In the original papers we considered a particular (rather cautious) dynamic preferences relation built into the semantics of the language. In this approach we tried to avoid allowing ambiguity in the specification of preferences, preferring inconsistency of a program to obtaining results not necessarily meant to by the program designer. It seems that for some applications this was a good idea, while for others it was clearly inadequate. This is, of course, a general problem of representing preferences which, in my judgment, is not yet sufficiently understood (despite very substantial progress in this area). There are other languages which are somewhat similar to CR-Prolog. The most similar probably are [25] and [10]. The first introduced a preference relation on rules. The second formalized weak constraints implemented in DLV. Unfortunately, none of these languages has a "final solution" to the problem of preferences. This is an important topic for further research.

# 5 Recent work on extensions of ASP

Finally, I'd like to briefly mention some of my more recent work on expanding ASP. In [7, 8] my colleagues and I introduced a new KR language P-log. Our goal was to create a language which would

- allow elegant formalizations of non-trivial combinations of logical and probabilistic reasoning,
- help the language designers (and hopefully others) to better understand the meaning of probability and probabilistic reasoning,

- help to design and implement knowledge-based software systems.

The logic part of P-Log is based on ASP (or its variants such as CR-Prolog). On the probabilistic side, we adopted the view which understands probabilistic reasoning as *commonsense reasoning about degrees of belief of a rational agent*. This matches well with the ASP-based logic side of the language. The ASP part of a P-log program can be used for describing possible beliefs, while the probabilistic part would allow knowledge engineers to quantify the degrees of these beliefs. Another important influence on the design of P-log is the separation between *doing* and *observing* and the notion of *Causal Bayesian Net* (see [36]). The language has a number of other attractive features which do not normally occur in probability theory. P-log probabilities are defined with respect to an explicitly stated knowledge base. In addition to having logical non-monotonicity, P-log is "probabilistically non-monotonic" — new information can add new possible worlds and change the original probabilistic model. Possible updates include defaults, rules introducing new terms, observations, and deliberate actions in the sense of Pearl. Even though much more research is needed to better understand mathematical properties of P-log and develop the methodology of its use, I am confident that we have already succeeded in our first two goals. To satisfy the third goal we need to develop algorithms that truly combine recent advances in probabilistic reasoning with that of ASP — a non-trivial and fascinating task.

The second extension of ASP addresses a computational bottleneck of "classical" ASP solvers. These solvers are based on grounding — the process which replaces rules with variables by the sets of their ground instantiations. If a program contains variables ranging over large (usually numerical) domains, its ground instantiation can be huge (despite the best efforts of intelligent grounding algorithms employed by such solvers). This causes both memory and time problems and makes ASP solvers practically useless for a given task. In [9] we suggested an extension of the standard ASP syntax. The new syntactic construct facilitates the creation of a new type of ASP solver which partially avoids grounding and combines standard ASP reasoning techniques with that used by constraint logic programming algorithms. The work was extended in a number of different directions (see for instance [29, 13, 16]) which substantially broadened the scope of applicability of the ASP paradigm.

# 6 Conclusion

This paper contains several examples of applying a particular methodology of research in knowledge representation and reasoning to the design of KR languages. In all these examples the research started with a theoretical question aimed at understanding some form of reasoning. Attempts to solve such a question normally lead to the design of a KR language which was then evaluated on its ability to model basic conceptual notions, faithfulness to intuition, and mathematical accuracy and elegance. At this stage the language could be of sub-

stantial interest to a logician who tries to understand correct modes of thinking. This first step is usually followed by the design and implementation of a naive reasoning engine which is used for the development of small experimental systems capable of performing carefully selected, intelligent tasks. This allows the researcher to find and tune proper knowledge representation methodology and to test the ability of the language to guide the design. At this stage the work may become even more interesting to the logician, but may also have a practical use as a specification language for a software designer. If the researchers want to use the language as an executable specification for real applications, they may need to spend a very substantial amount of time on discovering and implementing non-trivial reasoning algorithms, learning the application domain, and getting involved in a serious knowledge representation project. This type of experimental engineering not only shows practicality of the approach, but also helps a researcher to collect valuable feedback which can be used in the continuous process of improvement and extension. Even though at different points in time I was involved in work on most of these stages, I believe that, as a rule, they are quite different in nature, require different talents, and are sometimes impossible without a well assembled group of people. The clear realization of this fact by the community may help researchers to successfully publish their work even if it does not contain experimental results comparing efficiency of different implementations or non-trivial mathematical theorems establishing practically important properties of known languages. Finally, it may be interesting to add that the development of the above methodology was guided by the desire to satisfy the first four criteria for the knowledge representation language from section 2. *I found another four criteria substantially less important and sometimes even harmful.*

Overall I believe that the ASP research program of which I was a small part has very impressive accomplishments. We better understand the intuition behind such basic notions as belief, defaults, probability, causality, intentions, etc. This was done by the development of new mathematical theory and the methodology of its applications, and by experimental engineering. This foundational work helped to put our science on solid ground.

We are learning how to use our theories to build transparent, elaboration tolerant, provably correct, and efficient software systems. Twenty years ago I didn't believe that such systems would be possible in my lifetime. I am obviously happy to be proved wrong. For me, this work has been, and (I hope) will continue to be, deeply satisfying. I also hope that it will be equally satisfying for new generations of researchers.

# References

[1] K. Apt, A. Blair, and A. Walker. *Towards a theory of declarative knowledge*, pages 89–148. Foundations of deductive databases and logic programming. Morgan Kaufmann, 1988.

[2] Marcello Balduccini. Cr-models: An inference engine for cr-prolog. In *International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR-7*, pages 18–30, Jan 2004.

[3] Marcello Balduccini. USA-Smart: Improving the Quality of Plans in Answer Set Planning. In *PADL'04*, Lecture Notes in Artificial Intelligence (LNCS), Jun 2004.

[4] Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)*, 3(4–5):425–461, Jul 2003.

[5] Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In Patrick Doherty, John McCarthy, and Mary-Anne Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, pages 9–18, Mar 2003.

[6] Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19(20):73–148, 1994.

[7] Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic Reasoning with Answer Sets. In *Proceedings of LPNMR-7*, Jan 2004.

[8] Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. *Journal of Theory and Practice of Logic Programming (TPLP)*, 9(1):57–144, 2009.

[9] Baselice, P.A. Bonatti, and Michael Gelfond. Towards an integration of answer set and constraint solving. In *Proceedings of ICLP-05*, pages 52–66, 2005.

[10] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and weak constraints in Disjunctive Datalog. In *Proceedings of LPNMR-4*, pages 2–17, 1997.

[11] Weidong Chen and David S Warren. Computation of stable models and its integration with logical query processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

[12] K. Clark. Negation as failure. Logic and Data Bases, pages 293–322. Plenum Press, 1978.

[13] Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors. *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*. Springer, 2009.

[14] Chris Evans. Negation-as-failure as an approach to the Hanks and McDermott problem. In *Proceedings Second Int'l Symp. on Artificial Intelligence*, 1989.

[15] Terracina G., Leone N., Lio V., and Panetta C. Experimenting with recursive queries in database and logic programming systems. *Journal of Theory and Practice of Logic Programming (TPLP)*, 8(2):129–165, 2008.

[16] Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In *ICLP*, pages 235–249, 2009.

[17] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[18] Michael Gelfond. On stratified autoepistemic theories. In *Proceedings of AAAI87*, pages 207–211, 1987.

[19] Michael Gelfond. Autoepistemic logic and formalization of commonsense reasoning. In *Nonmonotonic Reasoning*, volume 346 of *Lecture Notes in Artificial Intelligence*, pages 176–187, 1989.

[20] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.

[21] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.

[22] Michael Gelfond and Vladimir Lifschitz. Representing Action and Change by Logic Programs. *Journal of Logic Programming*, 17(2–4):301–321, 1993.

[23] Michael Gelfond and Tran Cao Son. Reasoning with Prioritized Defaults. In *Third International Workshop, LPKR'97*, volume 1471 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 164–224, Oct 1997.

[24] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence Journal*, 33(3):379–412, Nov 1987.

[25] Katsumi Inoue and Chiaki Sakama. Representing Priorities in Logic Programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, pages 82–96. MIT Press, 1996.

[26] Victor W. Marek and Miroslaw Truszczynski. *Stable models and an alternative logic programming paradigm*, pages 375–398. The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin, 1999.

[27] Norman McCain and Hudson Turner. A causal theory of ramifications and qualifications. *Artificial Intelligence*, 32:57–95, 1995.

[28] John McCarthy. *Formalization of common sense, papers by John McCarthy edited by V. Lifschitz.* Ablex, 1990.

[29] Veena S. Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Ann. Math. Artif. Intell.*, 53(1-4):251–287, 2008.

[30] Jack Minker. On indefinite data bases and the closed world assumption. In *Proceedings of CADE-82*, pages 292–308, 1982.

[31] Robert C. Moore. Semantical considerations on nonmonotonic logic. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 272–279. Morgan Kaufmann, Aug 1983.

[32] Paul Morris. The anomalous extension problem in default reasoning. *Artificial Intelligence Journal*, 35(3):383–399, 1988.

[33] Leone N., Pfeifer G., Faber W., Eiter T., Gottlob G., Perri S., and Scarcello F. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2006.

[34] Ilkka Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, Jun 1998.

[35] Ilkka Niemela and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 420–429, 1997.

[36] J. Pearl. *Causality*. Cambridge University Press, 2000.

[37] Raymond Reiter. *On Closed World Data Bases*, pages 119–140. Logic and Data Bases. Plenum Press, 1978.

[38] Raymond Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.

[39] V Subrahmanian and C. Zaniolo. Relating stable models and AI planning domains. In *Proceedings of ICLP-95*, pages 233–247. 1995.

[40] Bonnie Traylor and Michael Gelfond. Representing null values in logic programming. In *Proceedings of LFCS'94*, pages 341–352. 1994.

[41] Richard Watson. An application of action theory to the space shuttle. In *Proc. First Int'l Workshop on Practical Aspects of Declarative Languages (Lecture Notes in Computer Science 1551*, pages 290–304. Springer-Verlag, 1998.