Eleventh Mexican International Conference on Artificial Intelligence: Advances in Artifical Intelligence and Applications, Special Session - Revised Papers

Implementing a Knowledge Bases Debugger

Juan C. Acosta Guadarrama, J. Raymundo Marcial Romero, Marcelo Romero
Computer Science Department
Autonomous University of Mexico State, FI-UAEM
Mexico
jguadarrama@gmail.com

Jorge Hernández Camacho
Computer Science Department
Autonomous University of Hidalgo State, ESH
Mexico
jhcjorge@gmail.com

***Abstract*—Knowledge representation is an important topic in common-sense reasoning and Artificial Intelligence, and one of the earliest techniques to represent it is by means of knowledge bases encoded into logic clauses. Encoding knowledge, however, is prone to typos and other kinds of consistency mistakes, which may yield incorrect results or even internal contradictions with conflicting information from other parts of the same code. In order to overcome such situations, we propose a logic-programming system to debug knowledge bases. The system has a strong theoretical framework on knowledge representation and reasoning, and a suggested on-line prototype where one can test logic programs. Such logic programs may have, of course, conflicting information and the system shall prompt the user where the possible source of conflict is. Besides, the system can be employed to identify conflicts of the knowledge base itself and upcoming new information, it can also be used to locate the source of conflict from a given inherently inconsistent static knowledge base. This paper describes an implementation of a declarative version of the system that has been characterised to debug knowledge bases in a semantical formalism. Some of the key components of such implementation are existing solvers, so this paper focuses on how to use them and why they work, towards an implemented a fully-fledged system.**

***Keywords*-Answer Set Programming; Knowledge Representation; program transformation; implementation; non-monotonic reasoning; belief revision**

ACKNOWLEDGMENT

This project has been supported by a grant from the Mexican Public-Education Ministry, SEP.

I. INTRODUCTION

As one of the major and traditional topics of artificial intelligence over the last years, knowledge representation and commonsense reasoning have proved to be strong theoretical frameworks for logic programming (LP) to manage knowledge bases. As a result, LP has become more widely applied in the administration knowledge bases of intelligent (rational) agents, especially when talking about an agent's incomplete knowledge in a changing environment. Such knowledge typically is encoded into a traditional LP language like `Prolog` or ASP and the resulting program is said to be the knowledge base of, say, from an intelligent agent to a logical circuit.

Although encoding knowledge into a logic program is a very important topic of research over the last few decades, little attention has been paid to the problem of debugging the resulting knowledge bases and how to implement a solution. The problem arises when having a typo or another kind of mistake while encoding, and it can be a gross problem if the knowledge base is particularly large.

Several authors [1]–[7] propose methods to manage knowledge bases and use *Answer Set Programming* [8]—or simply ASP—as their foundation for being one of the most solid and studied semantics for logic programs over the last decade. The approaches and the semantics itself, however, cannot provide an explicit method to self-debug its programs when conflicting rules exist or one needs to incorporate new information that conflicts with previous knowledge. For example, let us, consider the following simple knowledge base.

Example 1: Suppose we have the following simple logic program (in ASP) representing the knowledge of an agent, which acts under specific circumstances.

$$\begin{aligned} \mathcal{P} = \{ sleep &\leftarrow night, \neg tv(on) \\ night &\leftarrow \top \\ watch(tv) &\leftarrow tv(on) \\ tv(on) &\leftarrow \top \} \end{aligned}$$

The unique model of such program is $\{night, watch(tv), tv(on)\}$. Now suppose that for some reason, we need to incorporate new information encoded into the rule $\sim watch(tv) \leftarrow \top$. One can easily realise that the sole inclusion of the new information into the initial knowledge base,

$$\begin{aligned} \mathcal{P}' = \{ sleep &\leftarrow night, \neg tv(on) \\ night &\leftarrow \top \\ watch(tv) &\leftarrow tv(on) \\ tv(on) &\leftarrow \top \\ \sim watch(tv) &\leftarrow \top \} \end{aligned}$$

has no models either. What is more, the lack of an appropriate conclusion has collapsed the entire knowledge base and

978-0-7695-4904-0/12 $26.00 © 2012 IEEE
DOI 10.1109/MICAI.2012.20

no further conclusions can be drawn from it. So, the agent knows nothing at all now.

Of course, the above mentioned references can perform a corresponding update and automatically incorporate the new information into a new consistent knowledge base that does draw appropriate conclusions. Naturally the purpose of such references is a little different from the one we pose in this work. As a result, once the update is performed, one can get the resulting models, if any, and (in one of the references) the corresponding new knowledge bases. The problem with the approahes arises, however, when one tries to update an initial inconsistent knowledge base.

There are several earlier related works that have addressed the problem of debugging knowledge bases. For instance, [5] proposed a framework to restore consistency of a given list of planning specifications encoded into an ASP static[1] program. The proposal introduces the use of a method called *Consistency-Restoring Rules* to diagnose the source of conflict in a very particular context. Another proposal is in order when changing "factory" specifications of the knowledge base of an intelligent agent coping with a dynamic changing environment, by means of program transformations [7]. Although the latter is in the same context we suggest, they both have not been extended to the general case of debugging knowledge bases, besides having to move to the particular syntax they propose.

As a result, the present work introduces a general alternative to those proposals, and it addresses both kinds of problems into a single framework, and without need of a different language. We propose a prototype for debugging of knowledge bases by a more concise and precise method of relazation. In addition to that, this work introduces some formal specifications for the implementation on top of an ASP solver called DLV [9].

In this paper the reader can find a general description of a system to implement a debugger, as well as tools, methods, and directions to get the running solver itself. It begins with preliminary notation in Section II, followed by the main declarative definitions of the semantics and a couple of properties (Section III), which are the core of the paper together with some implementation techniques Section IV. The last section gives final conclusions and future work.

II. Preliminaries

This section is quite general and then we expect the reader to be familiar with basic notions of logic programming, Answer Sets Programming and non-monotonic reasoning from the literature.

A. Logic Programming and Answer Sets

The main base of our proposal is Answer Set Programming —ASP— [8] that is a well-known semantics for its logical properties and it is one of the foundations many authors propose for their approaches. Its formal language and some more notation are introduced *very* briefly as follows.

Definition 1 (Language $\mathcal{L}_{ASP}$ of logic programs): In the following we use the language of propositional logic with propositional symbols: $p_0, p_1, \ldots$; connectives: "," (conjunction); " |", " $\vee$" (disjunction); $\leftarrow$ (implication); $\bot$ (falsum); $\top$ (verum); "not", " $\neg$" (default negation); "$\sim$" (strong negation); auxiliary symbols: "(", ")" (parentheses). The propositional symbols are also called *atoms* or *atomic propositions*. A *literal* is an atom or an atom negated by $\sim$ A *rule* ρ is an ordered pair $\mathsf{Head}(\rho) \leftarrow \mathsf{Body}(\rho)$, where $\mathsf{Head}(\rho)$ is a (possibly empty) finite set of literals, and $\mathsf{Body}(\rho)$ a (possibly empty) finite set of (default-negated) literals.

The syntax of a logic programs can be defined as follows.

Definition 2 (EDLP): An *extended disjunctive logic program* is a set of rules of form

$$\ell_1 \vee \ell_2 \vee \ldots \vee \ell_l \leftarrow \ell_{l+1}, \ldots, \ell_m, \neg\ell_{m+1}, \ldots, \neg\ell_n \quad (1)$$

where ℓ_i is a literal and $0 \leq l \leq m \leq n$.

Naturally, an *extended logic program* (or ELP hereafter) is a finite set of rules of form (1) with $l = 1$; while an *integrity constraint* (also known in the literature as *strong constraint*) is a rule of form (1) with $l = 0$; while a *fact* is a rule of the same form with $l = m = n$.

Informally, the semantics of such programs consists of reducing the general rules to rules without default negation "$\neg$", because the latter are universally well understood. For page limitation, we just skip the formal definition of such reduct, which can be easily found in the literature. Finally, we weil use the alternative default-negation symbol not $\cdot$ to denote encoded DLV programs rather than generic ones.

A component of our solver, a weak constraint is a constraint that may be violated in order to establish priorities amongst models, which was introduced in [9] having the following syntax.

Definition 3 (Weak Constraint [9]): A *weak constraint* (wc) is an expression of the form

$$:\sim b_1, \ldots, b_k, \text{not } b_{k+1}, \ldots \text{not } b_m [w : l] \quad (2)$$

where for $0 \leq k \leq m$, $b_1, \ldots, b_m$ are literals, while w (the weight) and l (the level, or layer) are positive integer constants or variables. For convenience, w and/or l may be omitted and are set to 1 in such case.

Informally, the interpretation of an ASP program containing such rules consist in minimising the sum of weights of violated weak constraints in the highest priority level, and among them those which minimise the sum of weights of the violated weak constraints in the next lower level, and so on. For page-limit reasons, we skip the formal semantics, which is easily accessible in [9] and others.

[1] It is static in the sense that the thorough knowledge base endures no change, while the dynamic part is the generation of plans itself.

Although ASP is our main basis, we need a means to set up preferences amongst partial models, so that we may choose the ones according to general principles. One of such intermediate mechanisms was introduced as Abductive Logic Programming in [10] and is briefly presented in the following section.

B. Minimal Generalised Answer Sets

Minimal Generalised Answer Sets (MGAS) is one of the semantics to interpret abductive programs, which provides a more general and flexible semantics than standard ASP. Some of the main definitions are the following.

Definition 4 (Abductive Logic Program [10]): An abductive logic program is a pair $\langle \mathcal{P}, \mathcal{A} \rangle$ where $\mathcal{P}$ is an arbitrary program and $\mathcal{A}$ a set of literals, called abducibles.

A way to interpret an abductive program is by generalised answer sets, as formally expressed in the following definition.

Definition 5 (Generalised Answer Sets GAS [10]): $M(\Delta)$ is a generalised answer set of the abductive program $\langle \mathcal{P}, \mathcal{A} \rangle$ if and only if $\Delta \subseteq \mathcal{A}$ and $M(\Delta)$ is an answer set of $\mathcal{P} \cup \{H \leftarrow \top \mid H \in \Delta\}$.

Once we get more than one generalised answer set, a preferred order can be established over their set inclusion. Accordingly, we can talk about a *Minimal Generalised Answer Set*—MGAS.

Definition 6 (Abductive Inclusion Order [10]): An order over generalised answer sets is as follows: Let $M(\Delta_1)$ and $M(\Delta_2)$ be generalised answer sets of $\langle \mathcal{P}, \mathcal{A} \rangle$, we define $M(\Delta_1) \leq_{\mathcal{A}} M(\Delta_2)$ if and only if $\Delta_1 \subseteq \Delta_2$.

III. Debugging Logic Programs

Once we have the necessary background we can introduce our proposal to implement the debugger. To begin with, we say a program is *consistent* when it has answer sets, and it is *inconsistent* when it is not consistent. Additionally, it is said to be *debuggable* in at least two cases. On the one hand, it is debuggable when it is originally consistent and we want to incorporate a new rule that yields an inconsistency. On the other hand, when it is originally inconsistent and we wish to make it consistent.

This should be clearer after the following two examples.

Example 2: Suppose the consistent ASP program

$$\mathcal{P} = \{(a \leftarrow \top), (b \leftarrow \text{not}\, c), (d \leftarrow \top)\}$$

to which we would like to incorporate a new rule: $(\sim b \leftarrow \text{not}\, e)$. As a result, we get the new inconsistent program

$$\mathcal{P}' = \{(a \leftarrow \top), (b \leftarrow \text{not}\, c), (d \leftarrow \top), (\sim b \leftarrow \text{not}\, e)\}$$

which thus is debuggable. Informally one can easily realise that the new rule and rule $(b \leftarrow \text{not}\, c)$ are the source of conflict.

As opposed to Example 2, where the new rule might be considered a formal update to $\mathcal{P}$, one might also want to debug a single (*static*) ELP, as in the following example.

Example 3: Suppose the inconsistent program

$$\{(a \leftarrow \top), (b \leftarrow \text{not}\, c), (d \leftarrow \top), (\sim b \leftarrow \top), (\sim a \leftarrow \text{not}\, e)\},$$

which has no answer sets. As a result, it is debuggable and one can easily realise that the following pairs of rules are in conflict:

$(a \leftarrow \top)$ with $(\sim a \leftarrow \text{not}\, e)$ and $(b \leftarrow \text{not}\, c)$ with $(\sim b \leftarrow \top)$.

In order to overcome situations like the two cases above, we propose a semantics that can deal with them and can locate and show the sources of conflict so that the inconsistency may be resolved at a meta-semantics.

The following framework shall debug the conflicting rules from an inconsistency and shall yield one or more transformed programs that are no longer inconsistent. Informally, the method consists in relaxing the program to be debugged, by means of a set of new default-negated distinguished literals that do not appear in the original language. Once relaxed, they are part of the abductive logic program that has minimal generalised answer sets, which can point out the conflicting rules.

Formally, an *α-relaxed rule* is a rule ρ that is weakened by a default-negated atom α in its body: $\text{Head}(\rho) \leftarrow \text{Body}(\rho) \cup \{\neg\alpha\}$. In addition, an *$\alpha$-relaxed program* is a set of α-relaxed rules. Finally, there is a particular case of ELP from Definition 1 that contains facts-only, defined as generalised program:

A *generalised program* of $\mathcal{A}$ is a set of rules of form $\{\ell \leftarrow \top \mid \ell \in \mathcal{A}\}$, where $\mathcal{A}$ is a given set of literals.

Definition 7 (Debug Program): Given a debug pair of extended logic programs, $\mathcal{P}_1 \uplus \mathcal{P}_2$, over a set of atoms $\mathcal{A}$ and a set of distinguished abducibles, $\mathcal{A}^*$, where $\mathcal{A} \cap \mathcal{A}^* = \emptyset$, its *debugged program* is $\mathcal{P}' \cup \mathcal{P}_2 \cup \mathcal{P}_G$, where $\mathcal{P}'$ is the relaxed program for $\mathcal{P}$; $\mathcal{P}_G$ is a generalised program for $\mathsf{M} \cap \mathcal{A}^*$ for some *minimal generalised answer set* M of $\langle \mathcal{P}'_1 \cup \mathcal{P}_2, \mathcal{A}^* \rangle$ and "$\uplus$" is the corresponding debug operator.

A debug program has a corresponding model called *debug model*:

Definition 8 (Debug Model): Given a debug pair $\mathcal{P}_{\uplus} = \mathcal{P}_1 \uplus \mathcal{P}_2$ of extended logic programs over a set of atoms $\mathcal{A}$, the set $\mathcal{S} \subseteq \mathcal{A}$ is a *debug model of $\mathcal{P}_{\uplus}$* if and only if $\mathcal{S} = \mathcal{S}' \cap \mathcal{A}$ for some *minimal generalised answer set* $\mathcal{S}'$ of the corresponding abductive program.

A rule ρ is said to *contradict* a consistent ELP, $\mathcal{P} \setminus \{\rho\}$, when $\{\rho\} \cup \mathcal{P}$ has no answer sets.

Proposition 1: Suppose two consistent ELP's $\mathcal{P}_1$, $\mathcal{P}_2$ and a given debug pair $\mathcal{P}_{\uplus} = \mathcal{P}_1 \uplus \mathcal{P}_2$ such that $\mathcal{P}_1 \cup \mathcal{P}_2$ has no answer set. The rule $\rho \in \mathcal{P}_1$ contradicts $\mathcal{P}_2$ if and only if $\rho' \in \mathcal{P}'_1$, where ρ' is the corresponding α-relaxed rule

of ρ; $\mathcal{P}'$ is the relaxed program of $\mathcal{P}$; $\alpha \in \mathsf{M}$ and $\neg\alpha \in \mathsf{Body}(\rho')$, for some minimal generalised answer set M of its corresponding abductive program.

Proof sketch: The proof comes from the fact that $\mathcal{P}_2$ is consistent and an α-relaxed rule $\rho' \in \mathcal{P}_1'$ is inhibited when its corresponding abducible α is true. ■

In consequence, the α-rules in the debug program give enough information so as to find the source of conflict when one of them is not satisfied. So, one can always find contradictory rules as in the following example.

Example 4: From Example 2, one can suppose the new rule is in $\mathcal{P}_2$. So, rule $\rho = (b \leftarrow \text{not}\, c)$ contradicts $\mathcal{P}_2$ because the update pair is $\mathcal{P} \uplus \mathcal{P}_2$, which has a corresponding relaxed program $\mathcal{P}'$ and a corresponding relaxed rule of ρ, ρ', and the abductive program $\langle \mathcal{P}' \cup \mathcal{P}_2, \mathcal{A}^* \rangle$, with a MGAS, M, where $\alpha \in \mathsf{M}$ and $\text{not}\,\alpha \in \mathsf{Body}(\rho')$. As a result, α points out rule ρ as a source of conflict and the entire ρ' is not satisfied.

By recalling Example 3, one can also debug a single (static) ELP as follows.

Example 5: Given the inconsistent program $\mathcal{P}$ from Example 3, the debug pair corresponds to $\mathcal{P} \uplus \emptyset$ and the following pairs of rules $(a \leftarrow \top)$ with $(\sim a \leftarrow \text{not}\, x)$ and $(b \leftarrow \text{not}\, c)$ with $(\sim b \leftarrow \top)$ contradict each other for similar reasons.

In general, the following property holds when restoring consistency from an inconsistent program.

Proposition 2: Given an inconsistent ELP, $\mathcal{P}$, and the debug pair $\mathcal{P}_{\uplus} = \mathcal{P} \uplus \emptyset$. The rule $\rho \in \mathcal{P}$ contradicts $\mathcal{P}$ if and only if $\rho' \in \mathcal{P}'$, where ρ' is the corresponding α-relaxed rule of ρ, $\mathcal{P}'$ is the relaxed program of $\mathcal{P}$, $\alpha \in \mathsf{M}$ and $\neg\alpha \in \mathsf{Body}(\rho')$, for some minimal generalised answer set M of the corresponding abductive program from $\mathcal{P}_{\uplus}$.

This section has been an introduction to some of the main results of this paper, which consist of a general characterisation of contradictory information in ASP and weak constraints. They provide a solid theoretical framework towards the implementation of a system in DLV to debug knowledge bases. Further theoretical results, however, both on the properties of DLV's weak constraints and on consistency are omitted due to page-limit restrictions.

IV. Implementation

Currently there are at least three major efficient solvers for ASP with a long background of implementation and research. Some of those are DLV [9], CLASP and SMODELS [11], and our system debugs ELP programs of such solvers.

One of the proposals [12] to implement updates in MGAS was a setting of preferred disjunctive logic programs in ODLP [13] and an implementation for pairs of programs. Their justification in [12] to use ODLP is the implemented solver called PSmodels[2] that is an extension to SMODELS [11] to compute preferred answer sets. Unfortunately, up to now there is no stable version and the current one (v. 2.26a) endures some few bugs[3]. Moreover, it is believed that DLV significantly outperforms SMODELS [9], not to mention that ODLP is such a colossal system that can do much more complex tasks than just minimal inclusion models.

As an alternative to PSmodels we propose the use of DLV with a minimal-set-inclusion function (that is not yet included in DLV) for their characteristics of generalised answer sets.

A. The Parser and Grammar

Differently from the implementation suggested in [12], which has parser embedded in its PHP[4] code, this new parser has been compiled in C and it should be portable to nearly any other platform. The advantage of having a UNIX command-line binary module is the ease to be plugged in to other modules so as to build a more complex application. So, we propose the following tokens instances for this debug solver to take part at the scanner phase.

NAME	[~-]?[[:alnum:]_]+
GETS	":-"
NOT	"not "
RULEEND	"."
CONJUNCT	","

Note that, although we say *pairs of programs*, the case of an inherently inconsistent program that we have analysed in Section IV is still in order, as we can stick to Proposition 2 to handle it with the same debug definitions.

As soon as Flex scans the text of a program pair, its output is passed on to Yacc, which gives a meaning to each correct structure of rules and a program pair that contain rules. In particular, the Yacc process specifies the grammar for update pairs introduced in previous sections. It is also responsible for relaxing each rule in the first program of the pair, with a new distinguished atom and establishes a *set-inclusion-preference relation* among such atoms. Last, this process is responsible of an error-checking mechanism that verifies the correctness of the program according to a simple BNF grammar bellow.

```
<pair> ::= '{'<program>'}''{'<program>'}'
<program>     ::=<program> <rule> |
<rule>        ::=<head> END
                | <head> GETS <body> END
<head>        ::=<POST> |
<POST>        ::=<LITERAL>
                | <POST> DISJUNCT <LITERAL>
<body>        ::=<PRE> |
<PRE>         ::=<LITERAL>
                | NOT <LITERAL>
                | <PRE> CONJUNCT <LITERAL>
                | <PRE> CONJUNCT NOT <LITERAL>
<LITERAL>     ::=NAME
                | NAME '(' NAME ',' NAME ')'
```

[2] http://www.tcs.hut.fi/Software/smodels/priority/

[3] Try to compute the preferred models of the a simple program like $\{a.\}$.

[4] This is a script language quite suitable for small processes of dynamic contents on web pages.

As mentioned before, for each rule that is analysed, the system appends a pair of new rules with *relaxing atoms* and a weak constraint, in the first program of the pair:

$$-\alpha_i \mid \alpha_i \quad \leftarrow \quad \top \tag{3}$$

$$:\sim \quad \alpha_i \ [1:1] \tag{4}$$

where i represents the i^{th} abducible α and the latter forming a $[weight : level]$ weak constraint, and $weight = level = 1$ in our case.

The intuition behind this formulation is to compute the GAS's of the abductive program by violating the minimal set inclusion of abducibles under weak constraints syntax.

Example 6: Suppose we would like to restore consistency to the following simple ASP program:

$$\mathcal{P} = \{(a \leftarrow b), (c \leftarrow a), (e \leftarrow \text{not}\, e), (\sim a \leftarrow \text{not}\, g), (b \leftarrow \top)\}$$

The corresponding abductive program is $\langle \mathcal{P}' \cup \emptyset, \mathcal{A}^* \rangle$ where

$$\mathcal{P}' = \{(a \leftarrow b, \text{not}\, \alpha_1), (c \leftarrow a, \text{not}\, \alpha_2), (e \leftarrow \text{not}\, e, \text{not}\, \alpha_3), (\sim a \leftarrow \text{not}\, g, \text{not}\, \alpha_4), (b \leftarrow \text{not}\, \alpha_5)\}$$

and $\mathcal{A}^* = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$. It is not hard to see that the GAS's of such a program are:

$$\{a, b, c, \alpha_3, \alpha_4\}, \{b, \alpha_1, \alpha_3, \sim a\}, \{\alpha_3, \sim a, \alpha_5\}$$

The abductive program is then transformed into a DLV's preference program ready to be interpreted as a cardinality preference or as a set-inclusion preference semantics:

```
a:- b, not alpha_1.
-alpha_1 | alpha_1.      ~ alpha_1. [1:1]
c:- a, not alpha_2.
-alpha_2 | alpha_2.      ~ alpha_2. [1:1]
e:- not e, not alpha_3.
-alpha_3 | alpha_3.      ~ alpha_3. [1:1]
-a:- not g, not alpha_4.
-alpha_4 | alpha_4.      ~ alpha_4. [1:1]
b:- not alpha_5.
-alpha_5 | alpha_5.      ~ alpha_5. [1:1]
```

Once there is such a *preference program* like in Example 6, DLV can interpret it as *preferred answer sets*, which would cardinality-prefer the abducibles of the mentioned GAS's. Although such semantics should have interesting applications, in this case it would not necessarily give us the minimal set of conflicting rules of the original $\mathcal{P}$. Instead, DLV should extend the weak-constraints semantics to an additional switch that can allow the user to choose between cardinality preference and *set inclusion preference*. That is to say, such switch would allow us to choose between those models that violate the least number of weak constraints and those that violate the minimal set inclusions of the *simple weak constraints* rewritten by the system. Suppose one can make a choice, like in PSmodels[5]. As a result, from the preference program DLV would choose the minimal set inclusion of abducibles, which actually correspond to the abducibles of its MGAS's

$$\{\sim a, b, \alpha_1, \alpha_3\}, \{a, b, c, \alpha_3, \alpha_4\}, \{\sim a, \alpha_3, \alpha_5\}$$

from program $\langle \mathcal{P}' \cup \emptyset, \mathcal{A}^* \rangle$.

According to the semantics introduced in Section III, the MGAS's shown above can give us enough information to start debugging program $\mathcal{P}$: They mean that rule $\rho \in \mathcal{P}'$ where $\alpha_1 \in \text{Body}(\rho)$ conflicts with $\mathcal{P}$. That is, rule $(a \leftarrow b, \text{not}\, \alpha_1) \in \mathcal{P}'$, which is the relaxed form of $(a \leftarrow b) \in \mathcal{P}$ should be considered when debugging the inconsistent program $\mathcal{P}$. Such is the case of the rest of the rules.

In addition, the case of α_5 is worth considering. Notice that either rule $\rho_5 \in \mathcal{P}'$, where $\alpha_5 \in \text{Body}(\rho_5)$, is in conflict with $\mathcal{P}$, or rule $\rho_1 \in \mathcal{P}'$, where $\alpha_1 \in \text{Body}(\rho_1)$, is, or rule $\rho_4 \in \mathcal{P}'$, where $\alpha_4 \in \text{Body}(\rho_4)$, is. That is because rule number five depends on the truth value of rule number one, which are also in conflict with rule number four. So, the *metalanguage* has three different minimal ways to make $\mathcal{P}$ consistent by changing one of these three rules in question, besides rule number three!

Finally, rule $\rho \in \mathcal{P}$ where $\alpha_3 \in \text{Body}(\rho)$ is common to all the MGAS's and has no counterpart. That is obvious because the rule is intrinsically inconsistent in ASP.

B. The User Interface

The user interface is a simple web page consisting of the display of the original pair, its transformation to abductive program and its generalised answer sets, as well as the result of interpreting such an abductive program under MGAS and the update answer sets of the pair, whenever DLV can prefer under *set-inclusion criterion*, as we have suggested in this paper. This is a *rapid-prototype technique* encoded into a UNIX script, with some simple sub-processes that filter in the needed text from the DLV's formatted output. Last, this main module is also responsible for dealing with the user interface in HTML, by getting the input pair into a text pane on a web page and processing it to display the output within a new web page.

1) The Abductive Program: The abductive program is indeed encoded into a preference relation of weak constraints. The relation consists of a relaxed ELP where each rule has its corresponding pair of *disjunctive abducibles* (3) and a weak constraint (4).

This simple process forms the triple rule at the parsing stage by keeping a counter for each abducible, which is displayed once a rule is recognised as valid: the relaxed rule, a disjunctive rule and a weak constraint. The reader should note that such weak constraints shall be interpreted under set-inclusion preference rather than the original cardinality preference.

[5]The reader should recall that we mentioned before that although PSmodels is a good alternative, it is unpractical for two reasons: the few bugs it endures and the heavy load it is for a computer.

2) Computing Conflicting Rules: Computing *conflicting rules* is a straightforward process that takes the abductive program from the previous process as an input and passes it on to DLV. The general intuition behind this solver is to apply the ASP-reduct of the input ELP that returns none or more conflicting rules with their corresponding debugged program(s).

V. Discussion and Future Work

We have presented general methods for rapid prototyping of logic programming and for further research on optimisation techniques. We have also implemented the declarative version of both a debug semantics and MGAS's. The system has been thought with strong emphasis in declarative programming, in just some fragments of procedural modules, in order to make it easily modifiable for particular frameworks and as an evidence to confirm claims of the original semantics here prototyped. Another of its highlights is its modularity and UNIX philosophy that allows it to be a web service and easily plugged in to other systems even without needing to download it. Moreover, its simple standard graphical user interface in HTML makes it very easy to use, compared to most of the typical ASP solvers implemented for command-line use.

Implementation is one of the main components of Logic Programming, which helps quickly understand it (for educational proposes and for a reliable comparison tool, for instance). It also helps spread methods and compute large knowledge bases for more complex applications and future frameworks.

Some future works are in order. One of the most interesting is the implementation of a module to get the minimal set inclusion in DLV. Next, the complexity of the system should be formalised. On the theoretical side, there is work to do on consistency topics and debugging problems.

Some final informal remarks on the complexity in advance, though, are with respect to the program transformation. The transformed program should grow exponentially with respect to the original initial program.

References

[1] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits, "On properties of update sequences based on causal rejection," *Theory and Practice of Logic Programming*, vol. 2, no. 6, pp. 711–767, 2002.

[2] C. Sakama and K. Inoue, "An abductive framework for computing knowledge base updates," *Theory and Practice of Logic Programming*, vol. 3, no. 6, pp. 671–715, 2003.

[3] Y. Zhang and N. Foo, "A unified framework for representing logic program updates," in *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005)*, M. M. Veloso and S. Kambhampati, Eds. Pittsburgh, Pennsylvania, USA: AAAI Press / The MIT Press, 2005, pp. 707–713.

[4] J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite, "The refined extension principle for semantics of dynamic logic programming," *Studia Logica*, vol. 79, no. 1, pp. 7–32, 2005.

[5] M. Balduccini and M. Gelfond, "Logic programs with consistency-restoring rules," in *Proceedings of the AAAI Spring 2003 Symposium*. Palo Alto, California: AAAI Press, 2003, pp. 9–18. [Online]. Available: citeseer.nj.nec.com/564147.html

[6] ——, "CR-prolog: Logic programs with CR-rules," Knowledge Representation Lab—Texas Tech University, Tech. Rep., August 2003.

[7] J. C. Acosta-Guadarrama, J. Arrazola, and M. Osorio, "Making belief revision with LUPS," in *XI International Conference on Computing*, J. H. S. Azuela and G. A. Figueroa, Eds., no. ISBN: 970-18-8590-2. México, D.F.: CIC-IPN, November 2002.

[8] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming," in *Logic Programming, Proceedings of the Fifth International Conference and Symposium ICLP/SLP*, R. A. Kowalski and K. A. Bowen, Eds. Seattle, Washington: MIT Press, 1988, pp. 1070–1080.

[9] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV system for knowledge representation and reasoning," *ACM Transactions on Computational Logic*, vol. 7, no. 3, pp. 499–562, 2006.

[10] A. C. Kakas and P. Mancarella, "Generalized Stable Models: A semantics for abduction." in *ECAI*, Stockholm, Sweden, 1990, pp. 385–391.

[11] I. Niemela and P. Simons, "Smodels—an implementation of the Stable Model and Well-Founded Semantics for normal logic programs," in *Proceedings of the 4th LPNMR ('97)*, ser. LNCS, vol. 1265. Dagstuhl Castle, Germany: Springer, 1997, pp. 420–429.

[12] F. Zacarías, M. Osorio, J. C. Acosta-Guadarrama, and J. Dix, "Updates in Answer Set Programming Based on Structural Properties," in *7th International Symposium on Logical Formalizations of Commonsense Reasoning*, S. McIlraith, P. Peppas, and M. Thielscher, Eds., Fakultät Informatik, TU-Dresden. Corfu, Greece: ISSN 1430-211X, May 2005, pp. 213–219. [Online]. Available: www.iccl.tu-dresden.de/announce/CommonSense-2005/zacarias.pdf

[13] G. Brewka, "Logic programming with ordered disjunction," in *Proceedings of the 18th National Conference on Artificial Intelligence, AAAI-2002*. Edmonton, Alberta, Canada: Morgan Kaufmann, 2002. [Online]. Available: citeseer.nj.nec.com/brewka02logic.html