Side-Effect Inspection for Decision Making

Luís Moniz Pereira and Alexandre Miguel Pinto

Abstract. In order to decide on the course of action to take, one may need to check for side-effects of the possible available preferred actions. In the context of abduction in Logic Programs, abducible literals may represent actions and assumptions in the declarative rules used to represent our knowledge about the world. Besides finding out which alternative sets of actions achieve the desired goals, it may be of interest to identify which of those abductive solutions would also render *true* side-effect literals relevant for the decision making process at hand, and which would render those side-effects *false*. After collecting all the alternative abductive solutions for achieving the goals it is possible to identify which particular actions influence inspected side-effect literals' truth-value.

To achieve this, we present the concept of Inspection Point in Abductive Logic Programs, and show how, by means of examples, one can employ it to investigate side-effects of interest (the *inspection points*) in order to help evaluate and decide among abductive solutions. We show how this type of reasoning requires a new mechanism, not provided by others already available. We furthermore show how to implement this new mechanism it on top of an already existing abduction solving system — ABDUAL — in a way that can be adopted by other systems too.

Keywords: Abduction, Side-Effects, Decision Making, Posteriori Preferences.

1 Introduction

In this paper we present a new decision-making-aid reasoning mechanism for abductive logic programs — the inspection points (IPs). The IPs permit the declarative

Luís Moniz Pereira and Alexandre Miguel Pinto
Centro de Inteligência Artificial (CENTRIA)
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
e-mail: {lmp,amp}@di.fct.unl.pt

K. Nakamatsu et al. (Eds.): New Advan. in Intel. Decision Techno., SCI 199, pp. 139–149.
springerlink.com

specification and implementation of more efficient abductive logic programming based decision-making agents. The efficiency comes from that using IPs allows the agent to selectively specify the relevant side-effect consequences of abductions instead of computing all possible abductions and all their consequences, only to subsequently having to filter out irrelevant abductions and to ignore irrelevant side-effects.

Typically, in a logic programming setting, consequences of abductions are computed by some forward-chaining mechanism. The problem with such an approach is the waste in time and computing resources that comes from computing *all* the consequences of the adopted abductions, and not just the consequences relevant for the task at hand. A kind of "selective forward chaining" is what would be desired. The IPs we present and implement efficiently exactly enact such forward chaining. This is accomplished by first selecting, in a top-down fashion, the rules necessary for forward propagation, from choices made, to the side-effects whose inspection is desired. Posterior preferences among alternative solutions can then take the observed side-effects into account.

We begin by presenting the motivation, and some background notation and definitions follow. The general problem of reasoning with logic programs is addressed in section 2; in particular, we take a look at the nature of backward and forward chaining and their relationship to query answering in an abductive framework. In section 3 we very briefly describe our implementation of the IPs.

Further elaboration on possibilities of use of IPs is sketched, and conclusions and future work close the paper.

1.1 Motivation

When faced with some situation where several alternative courses of actions are available a rational agent must decide and choose which action to take. *A priori* preferences can be applied before choosing in order to reduce the number of considerable possible actions curtailing the explosion of irrelevant combinations of choices, but still several (possibly exclusive) may remain available. To make the best possible informed decision, and commit to a course of action, the agent must be enabled to foresee the consequences of its actions and then prefer on the basis of those consequences (with *a posteriori* preferences). Choosing which set of consequences is most preferred corresponds to an implicit choice on restricting which course of action to take. But only the consequences relevant to the *a posteriori* preferences should be calculated: there are virtually infinitely many consequences of a given action, most of which are completely irrelevant to the preference-based decision making. Other consequences may be just predictions about the present state of the world, and observing whether they are verified can eliminate hypothetical scenarios where certain decisions would appear to make sense. Not all consequences are experimentally observable though, hence IPs may serve to focus on the ones that are, and thus guide the experimentation required to decide among competing hypothesis, as in medical diagnosis say. That is, IPs can be put to the service of sifting through competing explanations, prior to any acting but albeit in preparing and configuring

the context for it. In science, such decisive consequences are often know as "crucial" side-effects, because hopefully they can guarantee excluding untoward possibilities.

Computationally too, there are many advantages as well to preferring a posteriori, i.e. to enact preferences on the computed models, after the consequences of opting for one or another abducible are known, by means of inspection points that examine specific side-effects of abduction. The advantages of so proceeding stem largely from avoiding combinatory explosions of abductive solutions, by filtering both irrelevant as well as less preferred abducibles.

We code the agent's knowledge in the form of a Normal Logic Program (NLP), and its possible actions or hypotheses as abducibles. When trying to find an existential answer to a query where some desired goal is achieved, the answers will include the abductions produced at top-down query-answering time. Instead of computing whole models to find out the side-effect consequences of the abductions, we introduce — and use — the new mechanism of *inspection point* whereby we declaratively specify which side-effect consequences of the possible answers are of interest. In the usual abductive reasoning setting, normal top-down query answering resorts to performing abduction to construct hypothetical answers. However, when using inspection points we just want to check if some literal is a consequence of the abductions made when finding an answer to the query. Thus, further abduction is disabled when inspection point checking.

We show how this type of reasoning requires a new mechanism, not provided by others already available. We furthermore show how to implement this new mechanism on top of an already existing abduction solving system — ABDUAL [3] — in a way that can be adopted by other systems too.

Example 1. **Relevant and irrelevant side-effects.** Consider this logic program where $drink\_water$ and $drink\_beer$ are abducibles.

$\leftarrow thirsty, not\ drink.$ % This is an Integrity Constraint
$wet\_glass \leftarrow use\_glass.$ $use\_glass \leftarrow drink.$
$drink \leftarrow drink\_water.$ $drink \leftarrow drink\_beer.$
$thirsty.$ $drunk \leftarrow drink\_beer.$
$unsafe\_drive \leftarrow inspect(drunk).$

Suppose we want to satisfy the Integrity Constraint, and also to check if we get drunk or not. However, we do not care about the glass becoming wet — that being completely irrelevant to our current concern. In this case, full forward-chaining or computation of whole models is a waste of time, because we are interested only in a subset of the program's literals. What we need is a selective ersatz forward chaining mechanism, an inspection tool which permits to check the truth value of given literals as a consequence of the abductions made to satisfy a given query plus any Integrity Constraints.

Moreover, in this example, if we may simply want to know the side-effects of the possible actions in order to decide (to drive or not to drive) **after** we know which side-effects are true. In such case, we do not want to the IC $\leftarrow not\ unsafe\_drive$ because that would always impose $not\ drink\_beer$. We want to allow all possible

solutions for the single IC $\leftarrow thirsty, not\ drink$ and then check the side-effects of each abductive solution.

1.2 Background Notation and Definitions

Definition 1. Logic Rule. A (Normal) Logic Rule has the general form
$A \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$
where A and the B_i and C_j are atoms.

A is the head of the rule, and $B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ its body. 'not ' denotes default negation. When the body of a rule is empty, we say its head is a fact and write the rule simply as A.

Definition 2. Logic Program. A (Normal) Logic Program (LP for short) P is a (possibly infinite) set of Logic Rules, standing for all its ground instances.

Definition 3. Integrity Constraint. An Integrity Constraint (IC) is a logic rule, expressing a denial, whose head is the reserved atom '*false*'.

A simpler way of writing an IC is by omitting the head of its rule. An example is the first rule of the program in example 1, meaning that '$thirsty$' cannot be *true* whenever '$drink$' is *false*, and vice-versa.

In the next sections, we focus on abductive logic programs, i.e., those with abducibles. Abducibles are literals that are not defined by any rules and correspond to hypotheses that one can independently assume or not — apart from eventual Integrity Constraints. Abducibles or their default negations can appear in bodies of rules just like any other literal. They are specified along with the LP.

2 Reasoning with Logic Programs

Recall that when finding an abductive solution for a query, one may want to check whether some other literals become *true* or *false* strictly within the abductive solution found (i.e., whether they are consequences, or side-effects, of such abductions), but without performing additional abductions, and without having to produce a complete model to do so. This type of reasoning requires a new mechanism. To achieve it, we introduce the concept of inspection point, and show how one can employ it to investigate side-effects of interest. Procedurally, the checking of an inspection point corresponds to performing a top-down query-proof for the inspected literal, but with the specific proviso of disabling new abductions during that proof. The proof for the inspected literal will succeed only if the abducibles needed for it were already, or will be adopted, in the present ongoing solution search for the top query. Consequently, this check is performed after a solution for the query has been found. At inspection-point-top-down-proof-mode, whenever an abducible is encountered, instead of adopting it, we simply adopt the intention to *a posteriori* check if the abducible is part of the answer to the query (unless of course the negation of the

abducible has already been adopted by then, allowing for immediate failure at that search node.) That is, one (meta-)abduces the checking of some abducible A, and the check consists in confirming that A is part of the abductive solution by matching it with the object of the check. In our method, the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals subject to inspection mode within a reserved construct $inspect/1$.

2.1 Backward and Forward Chaining

Query-answering is intrinsically backward-chaining as it is a top-down dependency-graph oriented proof-procedure. And all the more so of typically by need abductive query-answering. Finding the side-effects of a set of assumptions is conceptually envisaged as forward-chaining as it consists of progressively deriving conclusions from the assumptions until the truth value of the chosen side-effect literals is determined. The problem with full-fledged forward-chaining is that too many (often irrelevant) conclusions are derived. Since efficiency is always a concern, wasting time and resources on deriving conclusions only to be discarded afterwards, is a flagrant setback. Even worse, in combinatorial problems, there may be many alternative solutions whose differences repose just on irrelevant conclusions. So, the unnecessary computation of irrelevant conclusions in full forward-chaining may be multiplied, leading to immense waste. A more rational solution, when one is focused on some specific conclusions of a set of premises, is afforded by selective top-down ersatz forward-chaining. In such a setting, ideally, the user would be allowed to specify the conclusions she is focused on, and only those would be computed in a backward-chaining fashion, checking whether they are consequences of desired abductions, but without abducing — hence their ersatz character. Combining backward-chaining with forward-chaining (and in particular with (ersatz) selective forward-chaining) allows for a greater precision in specifying what we wish to know, and altogether improve efficient use of computational resources.

Significantly, if abduction is enabled, the computation of side-effects should take place without further abduction, passively (though not destructively) just "consuming" abducibles that are "produced" elsewhere by abduction for the top query.

In the sequel, we shall show how such a selective forward chaining from a set of hypotheses can actually be prepared by backward chaining from the focused on conclusions — the inspection points — by virtue of a controlled form of abduction.

2.2 The Use of Stable Models

When we need to know the 2-valued truth value of all the literals in the program for the problem we are modeling and solving, the only solution is to produce complete models. In such a case, tools like *SModels* [13] or *DVL* [5] are adequate because they can indeed compute whole and all models for the (finitely grounded) program. Typically, each abducible is then coded as a pair of rules, that form an even loop over default negation between the abducible and a representation of its negation. One

may discuss other alternative semantics (2-valued and 3-valued) that can also be used in this situation, and compare them with Stable Models (SM) semantics [10]. In an abductive reasoning situation, however, computing the whole model entails pronouncement about each of the abducibles, whether or not they are relevant to the problem at hand, and subsequently filtering the irrelevant ones. When we simply want to find an answer to a query, we either compute a whole model and check if it entails the query (the way SM semantics does), or, if the underlying semantics we are using enjoys the *relevancy* property — which SM semantics do not — we can simply use a top-down proof-procedure (*à la* Prolog). In this second case, the user does not pay the price of computing a whole model, nor the price of abducing all possible abducibles or their negations, since the only abducibles considered will be those needed for answering the query.

2.3 Abduction

Abduction ([1, 2, 3, 6, 7, 8, 9, 11, 12]) can naturally be used in a top-down query-oriented proof-procedure to find an (abductive) answer to a query, where abducibles are leafs in the call-dependency graph. The Well-Founded Semantics (WFS), which enjoys relevancy, allows for abductive query answering. We used it in the implementation briefly described in section 3. Though WFS is 3-valued, the abduction mechanism it employs can be, and in our case is, 2-valued.

2.4 Inspection Points

From a practical perspective, under the abductive logic programming setting, a typical query with side-effect checking via IPs could look like $main\_query, inspect(checked\_side\_effect)$. If there is an answer to such query then we know that the abductions made to satisfy the $main\_query$ are sufficient to prove also the $checked\_side\_effect$.

2.4.1 Meta-abduction

Intuitively, meta-abduction, in this setting, consists in abducing the intention of *a posteriori* checking for the abduction of some abducible, i.e. the intention of verifying that the abducible is indeed adopted. In practice, when we want to meta-abduce some abducible 'x', we abduce a literal '$abduced(x)$', which represents the intention that 'x' is eventually abduced along the process of finding an answer. The check is performed after a complete abductive answer to the top query is found. Operationally, 'x' has been or will be abduced as part of the ongoing solution to the top goal. Meta-abduction can be implemented by any abduction capable system.

When using a system that allows only for the top-down dependency-graph-oriented abductive query-solving we must *simulate* this *selective* bottom-up *forward*

chaining by means of a top-down query where making actual (non-meta) abductions is disallowed. In this setting, *inspection points* are the literals whose truth value we are interested in.

Example 2. **Inspection Points.** Consider this NLP, where '$tear\_gas$', '$fire$', and '$water\_cannon$' are the only abducibles.

$\leftarrow police, riot, not\ contain.$	% this is an Integrity Constraint
$contain \leftarrow tear\_gas.$	$contain \leftarrow water\_cannon.$
$smoke \leftarrow fire.$	$smoke \leftarrow inspect(tear\_gas).$
$police.$	$riot.$

Notice the two rules for '$smoke$'. The first states that one explanation for smoke is fire, when assuming the hypothesis '$fire$'. The second states '$tear\_gas$' is also a possible explanation for smoke. However, the presence of tear gas is a much more unlikely situation than the presence of fire; after all, tear gas is only used by police to contain riots and that is truly an exceptional situation. Fires are much more common and spontaneous than riots. For this reason, '$fire$' is a much more plausible explanation for '$smoke$' and, therefore, in order to let the explanation for '$smoke$' be '$tear\_gas$', there must be a plausible reason — imposed by some other likely phenomenon. This is represented by $inspect(tear\_gas)$ instead of simply '$tear\_gas$'. The '$inspect$' construct disallows regular abduction — only meta-abduction — to be performed whilst trying to solve '$tear\_gas$'. I.e., if we take tear gas as an abductive solution for smoke, this rule imposes that the step where we abduce '$tear\_gas$' is performed elsewhere, not under the derivation tree for '$smoke$'. Thus, '$tear\_gas$' is an *inspection point*.

The Integrity Constraint, because there is '$police$' and a '$riot$', forces '$contain$' to be *true*, and hence, '$tear\_gas$' or '$water\_cannon$' or both, must be abduced. '$smoke$' is only explained if, at the end of the day, '$tear\_gas$' is abduced to enact containment.

Abductive solutions should be plausible. '$smoke$' is plausibly explained by '$tear\_gas$' if there is a reason, a best explanation, that makes the presence of tear gas plausible; in this case the riot and the police. Plausibility is an important concept in science which lends credibility to hypotheses. Assigning plausibility measures to situations is an issue orthogonal to the problem.

Example 3. **Nuclear Power Plant Decision Problem.** This example was extracted from [16] and adapted to our current issues. In this example the abducibles do not represent actions. In a nuclear power plant there is decision problem: cleaning staff can clean the power plant on cleaning days, but only if there is no sound alarm. The alarm sounds when the temperature in the main reactor rises above a certain threshold, or if the alarm itself is faulty. When the alarm sounds everybody must evacuate the power plant immediately! Abducible literals are $cleaning\_day$, $temperature\_rise$ and $faulty\_alarm$.

$$
\begin{array}{ll}
dust & \leftarrow cleaning\_day, inspect(not\ sound\_alarm) \\
sound\_alarm & \leftarrow temperature\_rise \\
sound\_alarm & \leftarrow faulty\_alarm \\
evacuate & \leftarrow sound\_alarm \\
 & \leftarrow not\ cleaning\_day
\end{array}
$$

Satisfying the unique IC imposes $cleaning\_day$ *true* and gives us three minimal abductive solutions: $S_1 = \{dust, cleaning\_day\}$,
$S_2 = \{cleaning\_day, sound\_alarm, temperature\_rise, evacuate\}$, and $S_3 = \{cleaning\_day, sound\_alarm, faulty\_alarm, evacuate\}$. If we pose the query $? - not\ dust$ we want to know what could justify to the cleaners dusting not to occur given that it is a cleaning day (this last is enforced by the IC). However, we do not want to abduce the rise in temperature of the reactor nor to abduce the alarm to be faulty in order to prove $not\ dust$. Any of these justifying two abductions must result as a side-effect of the need to explain something else, for instance the observation of the sounding of the alarm, expressed by the IC $\leftarrow not\ sound\_alarm$, which would then abduce one or both of those two abducibles as plausible explanations. The $inspect/1$ in the body of the rule for $dust$ prevents any abduction below $sound\_alarm$ to be made just to make $not\ dust$ true. One other possibility would be for the observations coded by ICs $\leftarrow not\ temperature\_rise$ or $\leftarrow not\ faulty\_alarm$ to be present in order for $not\ dust$ to be true as a side-effect. A similar argument can be made about evacuating: one thing is to explain why evacuation takes place, another altogether is to justify it as necessary side-effect of root explanations for the alarm to go off. These two pragmatic uses correspond to different queries: $? - evacuate$ and $? - inspect(evacuate)$, respectively.

A declarative semantics of Inspection Points is presented and detailed in `http://centria.fct.unl.pt/ lmp/publications/online-papers/IP08.pdf` but, due to lack of space, we omit it here.

3 Implementation

We based our practical work on a formally defined, implemented, tried and true abduction system: Abdual [3]. Meta-abduction is implemented adroitly by means of a new reserved abducible predicate which engages the abduction mechanism to try and discharge any meta-abductions by means of the corresponding abducible. The approach taken can easily be adopted by other abductive systems, as we had the occasion to check [4].

Abdual is composed of two modules: the preprocessor which transforms the original program by adding its dual rules, plus specific abduction-enabling rules; and a meta-interpreter allowing for top-down abductive query solving. When solving a query, abducibles are dealt with by means of extra rules the preprocessor added to that effect. These rules just add the name of the abducible to an ongoing list of current abductions, unless the negation of the abducible was added before to the list of false abducibles in order to ensure abduction consistency.

3.1 Abdual with Inspection Points

Inspection Points in Abdual function mainly by means of controlling the general abduction step, which involves very few changes, both in the pre-processor and the meta-interpreter. Whenever an '$inspect(X)$' literal is found in the body of a rule, where 'X' is a goal, a meta-abduction-specific counter — the '$inspect\_counter$' — is increased by one, in order to keep track of the allowed character, active or passive, of performed abductions. The top-down evaluation of the query for 'X' then proceeds normally. Actual abductions are only allowed if the counter is set to zero, otherwise only meta-abductions are allowed. After finding an abductive solution for the query 'X' the counter is decreased by one. Backtracking over counter assignations is duly accounted for.

Of course, this way of implementing the Inspection Points (with just one $inspect\_counter$) presupposes the abductive query answering process is carried out "depth-first", guaranteeing the order of the literals in the bodies of rules actually corresponds to the order they are processed.

The present implementation of Abdual with Inspection Points is available on request and is detailed in `http://centria.fct.unl.pt/~lmp/publications/online-papers/IP08.pdf`

In case the "depth-first" discipline is not followed, either because goal delaying is taking place, or multi-threading, or co-routining, or any other form of parallelism is being exploited, then each queried literal will need to carry its own list of ancestors with their individual $inspect\_counters$. This is necessary so as to have a means, in each literal, to know which and how many $inspect$s there are between the root node and the literal currently being processed, and which $inspect\_counter$ to update; otherwise there would be no way to determine if abductions or else meta-abductions should be performed.

An alternative implementation would rely instead not on ABDUAL but on XSB-prolog's XASP, to let Smodels produce partial models where abducibles and meta-abducibles are coded as even loops over default negation, as in ACORDA [14].

3.2 Comparing to Other Systems

We briefly compared our abduction system with inspection points to HyProlog [4].The HyProlog system supports abduction and additionally a system of assumptions, which differs from abducibles in that they are explicitly produced and explicitly applied. Creating an assumption in parallel with mentioning an abducible makes it possible, as Veronica Dahl showed us, to check the state of abducibles and thus provide a HyProlog implementation of inspection points.

In general, there is not such a big difference between the operational semantics of HyProlog and the Inspection Points implementation we present; however, there is a major functionality difference: in HyProlog we can only require consumption directly on abducibles, and with Inspection Points we can inspect any literal, not just abducibles. Moreover, the part of HyProlog concerned with abduction has a standard

first order semantics, whereas assumptions+expectations rely on a resource-oriented semantics *à la* linear logic.

4 Conclusions and Future Work

In the context of abductive logic programs, we have presented a new mechanism of inspecting literals, which corresponds to a selective forward chaining, that can be used to check for side-effects. The implementation of side-effects inspection, relying on the meta-abduction principle, which is at the heart of this contribution, permits consequence assessment which can be used to identify basic causality relationships usable for decision making. We have implemented the inspection mechanism within the Abdual [3] meta-interpreter and checked that it can easily be ported to other systems [4]. The semantics underlying Abdual (and, therefore, the current implementation of inspection points) is WFS with abduction. However, in case we need a total 2-valued semantics we need recourse to Layered Models [15] if we want to retain advantage of relevancy for top-down querying. Hence, our future work directions include extending the LM semantics with abduction, and, of course, meta-abduction. An efficient implementation of this semantics is also under way.

Acknowledgements. We thank Robert A. Kowalski and Verónica Dahl and Henning Christiansen for their insightful discussions and references to related works [4, 16].

References

1. Alferes, J., Leite, J., Pereira, L., Quaresma, P.: Planning as abductive updating. In: Kitchin, D. (ed.) Procs. of AISB 2000 (2000)
2. Alferes, J.J., Pereira, L.M., Swift, T.: Well-founded abduction via tabled dual programs. In: Intl. Conf. on Logic Programming, pp. 426–440 (1999)
3. Alferes, J.J., Pereira, L.M., Swift, T.: Abduction in well-founded semantics and generalized stable models via tabled dual programs. TPLP 4(4), 383–428 (2004)
4. Christiansen, H., Dahl, V.: Hyprolog: A new logic programming language with assumptions and abduction. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 159–173. Springer, Heidelberg (2005)
5. Citrigno, S., Eiter, T., Faber, W., Gottlob, G., Koch, C., Leone, N., Mateis, C., Pfeifer, G., Scarcello, F.: The dlv system: Model generator and advanced frontends (system description). In: Workshop Logische Programmierung (1997)
6. Console, L., Theseider Dupre, D., Torasso, P.: On the relationship between abduction and deduction. J. of Logic and Computation 1(5), 661–690 (1991)
7. Denecker, M., De Schreye, D.: Sldnfa: An abductive procedure for normal abductive programs. In: Apt (ed.) Procs. of the Joint Intl. Conf. and Symposium on Logic Programming, Washington, USA, pp. 686–700. The MIT Press, Cambridge (1992)
8. Dung, P.M.: Negations as hypotheses: An abductive foundation for logic programming. In: ICLP, pp. 3–17. MIT Press, Cambridge (1991)
9. Eiter, T., Gottlob, G., Leone, N.: Abduction from logic programs: semantics and complexity. Theoretical Computer Science 189(1–2), 129–177 (1997)

10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080. MIT Press, Cambridge (1988)
11. Inoue, K., Sakama, C.: A fixpoint characterization of abductive logic programs. Journal of Logic Programming 27(2), 107–136 (1996)
12. Kakas, A.C., Riguzzi, F.: Learning with abduction. In: Džeroski, S., Lavrač, N. (eds.) ILP 1997. LNCS, vol. 1297, pp. 181–188. Springer, Heidelberg (1997)
13. Niemelä, I., Simons, P.: Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 420–429. Springer, Heidelberg (1997)
14. Pereira, L.M., Lopes, G.: Prospective logic agents. In: Neves, J., Santos, M.F., Machado, J.M. (eds.) EPIA 2007. LNCS (LNAI), vol. 4874, pp. 73–86. Springer, Heidelberg (2007)
15. Pereira, L.M., Pinto, A.M.: Layered models top-down querying of normal logic programs. In: Proceedings of the Practical Aspects of Declarative Languages. LNCS. Springer, Heidelberg (to appear, 2009)
16. Sadri, F., Toni, F.: Abduction with negation as failure for active and reactive rules. In: Lamma, E., Mello, P. (eds.) AI*IA 1999. LNCS, vol. 1792, pp. 49–60. Springer, Heidelberg (2000)