Inspection Points and Meta-Abduction in Logic Programs

Luís Moniz Pereira and Alexandre Miguel Pinto
{lmp|amp}@di.fct.unl.pt

Centro de Inteligência Artificial (CENTRIA)
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

Abstract. In the context of abduction in Logic Programs, when finding an abductive solution for a query, one may want to check too whether some other literals become *true* (or *false*) as a consequence, strictly within the abductive solution found, that is without performing additional abductions, and without having to produce a complete model to do so. That is, such consequence literals may consume, but not produce, the abduced literals of the solution. We show how this type of reasoning requires a new mechanism, not provided by others already available. To achieve it, we present the concept of Inspection Point in Abductive Logic Programs, and show, by means of examples, how one can employ it to investigate side-effects of interest (the *inspection points*) in order to help choose among abductive solutions. The touchstone of enabling inspection points can be construed as meta-abduction, by (meta-)abducing an "abduction" to check (i.e. to passively verify) that a certain concrete abduction is indeed adopted in a purported abductive solution. We show how to implement inspection points on top of already existing abduction solving systems — ABDUAL and XSB-XASP — in a way that can be adopted by other systems too.
Keywords: Logic Programs, Abduction, Side-Effects.

1 Introduction

We begin by presenting the motivation, plus some background notation and definitions follow. Then issues of reasoning with logic programs are addressed in section 2, in particular, we take a look at abductive reasoning and the nature of backward and forward chaining and their relationship to query answering in an abductive framework. In section 3 we introduce inspection points, illustrate their need and their use with examples, and provide a declarative semantics. In section 4 we describe in detail our implementation of inspection points and illustrate its workings with an example. To close the paper we add conclusions, comparisons, and an elaboration on the possible use of inspection points in future work is sketched.

1.1 Motivation

Sometimes, besides needing to abductively discover which hypotheses to assume in order to satisfy some condition, we may also want to know some of the side-effects of those assumptions; in fact, this is rather a rational thing to do. But, most of the

time, we do not wish to know *all* possible side-effects of our assumptions, as some of them will be irrelevant to our concern. Likewise, the side-effects inherent in abductive explanations might not all be of interest. One application of abductive reasoning is that of finding which actions to perform, their names being coded as abducibles.

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
$wet\_glass \leftarrow use\_glass.$	$use\_glass \leftarrow drink.$
$drink \leftarrow drink\_water.$	$drink \leftarrow drink\_beer.$
$thirsty.$	$drunk \leftarrow drink\_beer.$
$unsafe\_drive \leftarrow drunk.$	

Suppose we want to satisfy the Integrity Constraint(IC), 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, computation of whole models is a waste of time, because we are interested only in a subset of the program's literals. Moreover, in this example, 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 a case, we do not want to simply introduce an IC expressed as $\leftarrow not\ unsafe\_drive$ because that would always impose abducing $not\ drink\_beer$. We want to allow all possible solutions for the single IC $\leftarrow thirsty, not\ drink$ and then check for the side-effects of each abductive solution.

What we need is an inspection mechanism which permits to check the truth value of given literals as a consequence of the abductions made to satisfy a given query plus any ICs, but without further abducing. This will be achieved just through the $inspect/1$ meta-predicate, by introducing the IC $\leftarrow inspect(not\ unsafe\_drive)$.

1.2 Background Notation and Definitions

Definition 1. ***Logic Rule.*** *A Logic Rule has the general form*
$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$
where H is an atom, and the B_i and C_j are atoms.

We call H the head of the rule, and $B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ its body. Throughout this paper we use 'not' to denote default negation. When the body of a rule is empty, we say its head is a fact and we write the rule just as H. When the head is empty, we designate the rule an Integrity Constraint (IC).

Definition 2. ***Logic Program.*** *A Logic Program (LP for short) P is a (possibly infinite) set of ground Logic Rules, where non-gound rules stand for all their ground instances.*

In this paper we consider solely Normal LPs (NLPs), those whose heads of rules are positive literals, i.e. simple atoms, or empty. In the next sections we focus on abductive logic programs, i.e., those with abducibles. Abducibles are chosen (by a system specific declaration) literals, not defined by any rules and correspond to hypotheses that one can independently assume or not — apart from eventual ICs affecting them. Abducibles or their default negations may appear in bodies of rules, just like any other literal.

2 Abductive Reasoning with Logic Programs

Logic Programs have been used for a few decades now in knowledge representation and reasoning. Amongst the most common kinds of reasoning performed using them, we can find deduction, induction and abduction. When query answering, if we know that the underlying semantics is *relevant*, i.e. guarantees it is enough to use only the rules relevant to the query (those in its call-graph) to assess its truthfulness, then we need not compute a whole model in order to find an answer to our query: it suffices to use just the call-graph relevant part of the program. This way of top-down finding a solution to a query, dubbed "backward chaining", is possible only when the underlying semantics is *relevant* in the above sense, because the existence of a full model is guaranteed.

Currently, the standard 2-valued semantics used by the logic programming community is Stable Models [8]. Its properties are well known and there are efficient implementations (such as *DLV* and *SModels* [3, 11]). However, Stable Models (SMs) miss some important properties, both from the theoretical and practical perspectives: guarantee of model existence for every NLP, relevancy and cumulativity. Since SMs do not enjoy relevancy they cannot use just backward chaining for query answering. This means that it may incur in waste of computational resources, when extra time and memory are used to compute parts of the model which may be irrelevant to the query.

When performing abductive reasoning, we want to find, by need only (via backward chaining), one possible set of conditions (abductive literals of the program to be assumed either *true* or *false*) sufficient to entail our query. However, sometimes we also want to know which are (some of) the consequences (side-effects, so to speak) of such conditions. I.e., we want to know the truth value of some other literals, not part of the query's call graph, whose truth-value may be determined by the abductive conditions found. In some cases, we might be interested in knowing every possible side-effect — the truth-value of every literal in a complete model satisfying the query and ICs. In other situations though, our focus is only in some specific side-effects of abductions performed.

In our approach, the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals within reserved construct $inspect/1$. It is advantageous, from a computational point of view, to be able to compute only the truth-value of the important side-effects instead of the whole model, so as not to waste precious time and computational resources. This is possible whenever the underlying semantics guarantees model existence, and enjoys relevance.

2.1 Abduction

Abduction, or inference to the best explanation, is a reasoning method whereby one chooses the hypotheses that would, if true, best explain the observed evidence. In LPs, abductive hypotheses (or *abducibles*) are named literals of the program which have no rules. They can be considered *true* or *false* for the purpose of answering a query. Abduction in LPs ([1, 4, 5, 9, 10]) 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) [7], which enjoys relevancy, allows for abductive query answering. We used it in the specific implementation

described in section 4 based on ABDUAL [1]. Though WFS is 3-valued, the abduction mechanism it employs can be, and in our case is, 2-valued.

Because they do not depend on any other literal in the program, abducibles can be modeled in a Logic Program system without specific abduction mechanisms by including for each abducible an even loop over default negation, e.g.,

$abducible \leftarrow not\ abducible\_not. \qquad abducible\_not \leftarrow not\ abducible.$

where $neg\_abducible$ is a new abducible atom, representing the (abducible) negation of the abducible. This way, under the SM semantics, a program may have models where some $abducible$ is *true* and another where it is *false*, i.e. $neg\_abducible$ is *true*. If there are n abducibles in the program, there will be 2^n models corresponding to all the possible combinations of *true* and *false* for each. Under the WFS without a specific abduction mechanism, e.g. the one available in ABDUAL, both $abducible$ and $neg\_abducible$ remain *undefined* in the Well-Founded Model (WFM), but may hold (as alternatives) in some Partial Stable Models.

Using the SM semantics abduction is done by guessing the truth-value of each abducible and providing the whole model and testing it for stability; whereas using the WFS with abduction, it can be performed *by need*, induced by the top-down query solving procedure, solely for the relevant abducibles — i.e., irrelevant abducibles are left unconsidered. Thus, top-down abductive query answering is a means of finding those abducible values one might commit to in order to satisfy a query.

An additional situation, addressed in this paper, is when one wishes to only passively determine which abducibles would be sufficient to satisfy some goal but without actually abducing them, just consuming other goals' needed and produced abductions. The difference is subtle but of importance, and it requires a new construct. Its mechanism, of *inspecting without abducing*, can be conceived and implemented through *meta-abduction*, and is discussed in detail in the sequel.

2.2 Backward and Forward Chaining

Abductive query-answering is intrinsically a backward-chaining process, a top-down dependency-graph oriented proof-procedure. Finding the side-effects of a set of abductive assumptions may be 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 of a model are derived. Wasting time and resources deriving them only to be discarded afterwards is a flagrant setback. Worse, there may be many alternative models satisfying an abductive query (and the ICs) whose differences just repose on irrelevant conclusions. So unnecessary computation of irrelevant conclusions can be compounded by the need to discard irrelevant alternative complete models too.

A more intelligent solution would be afforded by selective forward-chaining, where the user would be allowed to specify those conclusions she is focused on, and only those would be computed in forward-chaining fashion. Combining backward-chaining with selective forward-chaining would allow for a greater precision in specifying what we wish to know, and improve efficiency altogether. In the sequel we show how such a selective forward chaining from a set of abductive hypotheses can be replaced by

backward chaining from the focused on conclusions — the inspection points — by virtue of a controlled form of abduction which, never performing extra abductions, just checks for abducibles assumed elsewhere.

3 Inspection Points

Meta-abduction is used in *abduction inhibited inspection*. Intuitively, when an abducible is considered under mere inspection, meta-abduction abduces only the intention to *a posteriori* check for its abduction elsewhere, i.e. it abduces the intention of verifying that the abducible is indeed adopted, but elsewhere. In practice, when we want to meta-abduce some abducible 'x', we abduce a literal '$consume(x)$' (or '$abduced(x)$'), which represents the intention that 'x' is eventually abduced elsewhere in the process of finding an abductive solution. The check is performed after a complete abductive answer to the top query is found. Operationally, 'x' will already have been or will be later abduced as part of the ongoing solution to the top goal.

Example 2. **Police and Tear Gas Issue.** Consider this NLP, where '$tear\_gas$', '$fire$', and '$water\_cannon$' are the only abducibles. Notice that $inspect$ is applied to calls.

$\leftarrow police, riot, not\ contain.$	% 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 allowing 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 IC, 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, and '$smoke$' 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, for lending credibility to hypotheses. Assigning plausibility measures to situations is an orthogonal issue.

In this example, another way of viewing the need for the new mechanism embodied by the $inspect$ predicate is to consider we have 2 agents: one is a police officer and has the possibility of abducing (corresponding to actually throwing) $tear\_gas$; the other agent is a civilian who, obviously, does not have the possibility of abducing (throwing) $tear\_gas$. For the police officer agent, having the $smoke \leftarrow inspect(tear\_gas)$ rule, with the $inspect$ is unnecessary: the agent knows that $tear\_gas$ is the explanation for smoke because it was himself who abduced (threw) $tear\_gas$; but for the civilian agent the $inspect$ in the $smoke \leftarrow inspect(tear\_gas)$ rule is absolutely indispensable, since he cannot abduce $tear\_gas$ and therefore cannot know, without $inspect$ing, if that is the real explanation for $smoke$.

Example 3. **Nuclear Power Plant Decision Problem.** This example was extracted from [12] and adapted to our current designs, and its abducibles do not represent actions. In a nuclear power plant there is decision problem: cleaning staff will dust the power plant on cleaning days, but only if there is no alarm sounding. 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 the cleaners dusting not to occur given that it is a cleaning day (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, expressible by adding 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 two 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.

3.1 Declarative Semantics of Inspection Points

A simple transformation maps programs with inspection points into programs without them. Mark that the Stable Models of the transformed program where each $abducible(X)$ is matched by the abducible X (X being a literal a or its default negation $not\ a$) clearly correspond to the intended procedural meanings ascribed to the inspection points of the original program.

Definition 3. ***Transforming Inspection Points.*** *Let P be a program containing rules whose body possibly contains inspection points. The program $\Pi(P)$ consists of:*

1. *all the rules obtained by the rules in P by systematically replacing:*
   - $inspect(not\ L)$ *with* $not\ inspect(L)$*;*
   - $inspect(a)$ *or* $inspect(abduced(a))$ *with* $abduced(a)$ *if a is an abducible, and keeping $inspect(a)$ otherwise.*
2. *for every rule $A \leftarrow L_1, \ldots, L_t$ in P, the additional rule:* $inspect(A) \leftarrow L_1^{'}, \ldots, L_t^{'}$ *where for every $1 \leq i \leq t$:*

$$L_i^{'} = \begin{cases} abduced(L_i) & \text{if } L_i \text{ is an abducible} \\ inspect(X) & \text{if } L_i \text{ is } inspect(X) \\ inspect(L_i) & \text{otherwise} \end{cases}$$

The semantics of the $inspect$ predicate is exclusively given by the generated rules for $inspect$

Example 4. **Transforming a Program P with Nested Inspection Levels.**

$$\begin{array}{ll} x \leftarrow a, inspect(y), b, c, not\ d & y \leftarrow inspect(not\ a) \\ z \leftarrow d & y \leftarrow b, inspect(not\ z), c \end{array}$$

Then, $\Pi(P)$ is:

$$\begin{array}{ll} x & \leftarrow a, inspect(y), b, c, not\ d \\ inspect(x) & \leftarrow abduced(a), inspect(y), abduced(b), abduced(c), not\ abduced(d) \\ y & \leftarrow not\ inspect(a) \\ y & \leftarrow b, not\ inspect(z), c \\ inspect(y) & \leftarrow not\ abduced(a) \\ inspect(y) & \leftarrow abduced(b), not\ inspect(z), abduced(c) \\ z & \leftarrow d \\ inspect(z) & \leftarrow abduced(d) \end{array}$$

The abductive stable model of $\Pi(P)$ respecting the inspection points is:
$\{x, a, b, c, abduced(a), abduced(b), abduced(c), inspect(y)\}$.

Note that for each $abduced(a)$ the corresponding a is in the model.

4 Implementation

We based our practical work on a formally defined, XSB-implemented, true and tried abduction system — ABDUAL [1]. ABDUAL lays the foundations for efficiently computing queries over ground three-valued abductive frameworks for extended logic programs with integrity constraints, on the well-founded semantics and its partial stable models.

The query processing technique in ABDUAL relies on a mixture of program transformation and tabled evaluation. A transformation removes default negative literals (by making them positive) from both the program and the integrity rules. Specifically, a dual transformation is used, that defines for each objective literal O and its set of rules R, a dual set of rules whose conclusions $not\ (O)$ are true if and only if O is false in R. Tabled evaluation of the resulting program turns out to be much simpler than for the original program, whenever abduction over negation is needed. At the same time, termination and complexity properties of tabled evaluation of extended programs are preserved by the transformation, when abduction is not needed. Regarding tabled evaluation, ABDUAL is in line with SLG [13] evaluation, which computes queries to normal programs according to the well-founded semantics. To it, ABDUAL tabled evaluation adds mechanisms to handle abduction and deal with the dual programs.

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 lists failing in order to ensure abduction consistency. Meta-abduction is implemented adroitly by means of a reserved predicate, '$inspect/1$' taking some literal L as argument, which engages the abduction mechanism to try and discharge any meta-abductions performed under L by matching with the corresponding abducibles, adopted elsewhere outside any '$inspect/1$' call. The approach taken can easily be adopted by other abductive systems, as we had the occasion to check, e.g., with system [2]. We have also enacted an alternative implementation, relying on XSB-XASP and the declarative semantics transformation above, which is reported below.

Procedurally, in the ABDUAL implementation, 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 adopted, 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. According to our method, the side-effects of interest are explicitly indicated by the user by wrap-

ping the corresponding goals subject to inspection mode, with the reserved construct '$inspect/1$'.

4.1 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 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. We assume such a "depth-first" discipline in the implementation of inspection points, described in detail below. We lift this restriction at the end of the subsection.

Changes to the pre-processor:

1. A new dynamic predicate was added: the '$inspect\_counter/1$'. This is initialized to zero ('$inspect\_counter(0)$') via an assert, before a top-level query is launched.
2. The original rules for the normal abduction step are now preceded by an additional condition checking that the '$inspect\_counter$' is indeed set to zero.
3. Extra rules for the "inspection" abduction step are added, preceded by a condition checking the '$inspect\_counter$' is set to greater than zero. When these rules are called, the corresponding abducible 'A' is not abduced as it would happen in the original rules; instead, '$consume(A)$' (or '$abduced(A)$') is abduced. This corresponds to the meta-abduction: we abduce the need to abduce 'A', the need to 'consume' the abduction of 'A', which is finally checked when derivation for the very top goal is finished.

The changes to the meta-interpreter include all the remaining processing needed to correctly implement inspection points, namely matching the meta-abduction of '$consume(X)$' against the abduction of 'X'.

Changes to the meta-interpreter: The semantics we chose for the inspection points in ABDUAL is actually very close to that of the *deontic verifiers* mentioned before (and also below), in the sense that if a meta-abduction on 'X' (resulting from abducing '$consume(X)$') is not matched by an actual abduction on 'X' when we reach the end of solving the top query, the candidate abductive answer is considered invalid and the query solving fails. On backtracking, another alternative abductive solution (possibly with other meta-abductions) will be sought.

In detail, the changes to the meta-interpreter include:

1. Two 'quick-kill' rules for improved efficiency that detect and immediately solve trivial cases for meta-abduction:
   - When literal 'X' about to be meta-abduced ('$consume(X)$' about to be added to the abductions list) has actually been abduced already ('X' is in the abductions list) the meta-abduction succeeds immediately and '$consume(X)$' is not added to the abductions list;
   - When the situation in the previous point occurs, but with '$not\ X$' already abduced instead, the meta-abduction immediately fails.
2. Two new rules for the general case of meta-abduction, that now specifically treat the '$inspect(not\ X)$' and '$inspect(X)$' literals. In either rule, first we increase the '$inspect\_counter$' mentioned before, then proceed with the usual meta-interpretation for '$not\ X$' ('X', respectively), and, when this evaluation succeeds, we then decrease '$inspect\_counter$'.
3. After an abductive solution is found to the top query, check (impose) that every meta-abduction, i.e., every '$consume(X)$' literal abduced, is matched by a respective and consistent abduction, i.e., is matched by the abducible 'X' in the abductions list; otherwise the tentative solution found fails.

A counter — '$inspect\_counter$' — is used instead of a toggle because several '$inspect(X)$' literals may appear at different graph-depth levels under each other, and reseting a toggle after solving a lower-level meta-abduction would allow actual abductions under the higher-level meta-abduction. An example clarifies this.

Example 5. **Nested Inspection Points.** Consider again the program of the previous example, where the abducibles are a, b, c, d:

$$x \leftarrow a, inspect(y), b, c, not\ d. \qquad y \leftarrow inspect(not\ a).$$
$$z \leftarrow d. \qquad\qquad\qquad\qquad y \leftarrow b, inspect(not\ z), c.$$

When we want to find an abductive solution for x, skipping over the low-level technical details we proceed as follows:

1. a is an abducible and since the '$inspect\_counter$' is still set initially to 0 we can abduce a by adding it to the running abductions list;
2. y is not an abducible and so we cannot use any 'quick kill' rule on it. We increase the '$inspect\_counter$' — which now takes the value 1 — and proceed to find an abductive solution for y;
3. since the '$inspect\_counter$' is different from 0, only meta-abductions are allowed;
4. using the first rule for y we need to '$inspect(not\ a)$', but since we have already abduced a a 'quick-kill' is applicable here: we already know that this '$inspect(not\ a)$' will fail. The value of the '$inspect\_counter$' will remain 1;
5. on backtracking, the second rule for y is selected, and now we meta-abduce b by adding '$consume(b)$' to the ongoing abductions list;
6. increase '$inspect\_counter$' again, making it take the value 2, and continue on, searching an abductive solution for $not\ z$;
7. the only solution for $not\ z$ is by abducing $not\ d$, but since the '$inspect\_counter$' is greater than 0, we can only meta-abduce $not\ d$, i.e.,
   '$consume(not\ d)$' is added to the running abductions list;

8. returning to y's rule: the meta-interpretation of '$inspect(not\ z)$' succeeds and so we decrease the '$inspect\_counter$' by one — it takes the value 1 again. Now we proceed and try to solve c;
9. c is an abducible, but since the $inspect\_counter$ is set to 1, we only meta-abduce c by adding '$consume(c)$' to the running abductions list;
10. returning to x's rule: the meta-interpretation of '$inspect(y)$' succeeds and so we decrease the '$inspect\_counter$' once more, and it now takes the value 0. From this point onwards regular abductions will take place instead of meta-abductions;
11. we abduce b, c, and $not\ d$ by adding them to the abductions list;
12. a tentative abductive solution is found to the initial query. It consists of the abductions: $[a, consume(b), consume(not\ d), consume(c), b, c, not\ d]$;
13. the abductive solution is now checked for matches between meta-abductions and actual abductions. In this case, for every '$consume(A)$' in the abduction list there is an A also in the abduction list, i.e., every intention of abduction '$consume(A)$' is satisfied by the actual abduction of A. Because this final checking step succeeds, the whole answer is actually accepted. Note it is irrelevant which order a '$consume(A)$' and the corresponding A appear and were placed in the abductions list. The A in $consume(A)$ is just any abducible literal a or its default negation $not\ a$.

In this example, we can see clearly that the $inspect$ predicate can be used on any arbitrary literal, and not just on abducibles.

The correctness of this implementation against the declarative semantics provided before can be sketched by noticing that whenever the $inspect\_counter$ is set to 0 the meta-interpreter performs actual abduction which corresponds to the use of the original program rules; whenever the $inspect\_counter$ is set to some value greater than 0 the meta-interpreter just abduces $consume(A)$ (where A is the abducible being checked for its abduction being produced elsewhere), and this corresponds to the use of the program transformation rules for the $inspect$ predicate.

The implementation of ABDUAL with inspection points is available on request.

More general query solving 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 currently being processed literal, and which $inspect\_counter$ to update; otherwise there would be no way to know if abductions or meta-abductions should be performed.

4.2 Alternative Implementation Method

The method presented here is an avenue for implementing the inspection points mechanism through a simple syntactic transformation which can be readily used by any SMs system like SModels or DLV. Using an SMs implementation alone, one can get the abductive SMs of some program P by computing the SMs of P' where P' is obtained

from P by applying the program transformation we presented for the declarative semantics of the inspection points, and then adding an even loop over negation for each abducible (like the one depicted in section 2.1). Using XSB-Prolog's XSB-XASP interface, the process would be the same as for using an SMs implementation alone, but instead of sending the whole P' to the SMs engine, only the residual program, relevant for the query at hand, would be sent. This way, abductive reasoning can benefit from the relevance property enjoyed by the Well-Founded Semantics implemented by the XSB-Prolog's SLG-WAM.

Given the top-down proof procedure for abduction, implementing inspection points for program P becomes just a matter of adapting the evaluation of derivation subtrees falling under '$inspect/1$' literals, at meta-interpreter level, subsequent to performing the transformation $\Pi(P)$ presented before, which defines the declarative semantics. Basically, any considered abducibles evaluated under '$inspect/1$' subtrees, say A, are codified as '$abduced(A)$', where:

$$abduced(A) \leftarrow not\ abduced\_not(A)$$
$$abduced\_not(A) \leftarrow not\ abduced(A)$$

All $abduced/1$ literals collected during computation of the residual program are later checked against the stable models themselves. Every '$abduced(a)$' must pair with a corresponding abducible a for the model to be accepted.

5 Conclusions, Comparisons, and Future Work

In the context of abductive logic programs, we have presented a new mechanism of inspecting literals that can be used to check for side-effects, by relying on meta-abduction. We have implemented the inspection mechanism within the Abdual [1] meta-interpreter, and also in XSB-XASP, and checked that it can be ported to other systems [2].

HyProlog [2] is an abduction/assumption system which allows for the user to specify if an abducible is to be consumed only once or many times. In HyProlog, as the query solving proceeds, when abducibles/assumptions consumptions take place they are executed as storing the respective consumption intention in a store. After an abductive solution for a query is found, the actual abductions/assumptions are matched against the consumption intentions. 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.

In [12], the authors detect a problem with the IFF abductive proof procedure [6] of Fung and Kowalski, in what concerns the treatment of negated abducibles in integrity constraints (e.g. in their examples 2 and 3). They then specialize IFF to avoid such problems and prove correctness of the new procedure. The problems detected refers to the active use of an IC of some not A, where A is an abducible, whereas the intended use should be a passive one, simply checking whether A is proved in the abductive solution found. To that effect they replace such occurrences of not A by not provable(A), in order to ensure that no new abductions are allowed during the checking. Our own work

generalizes the scope of the problem they solved and solves the problems involved in this wider scope. For one we allow for passive checking not just of negated abducibles but also of positive ones, as well as passive checking of any literal, whether or not abducible, and allow also to single out which occurrences are passive or active. Thus, we can cater for both passive and active ICs, depending on the use desired. Our solution uses abduction itself to solve the problem, making it general for use in other abductive frameworks and procedures.

A future application of inspection points is planning in a multi-agent setting. An agent may have abduced a plan and, in the course of carrying out its abduced actions, it may find that another agent has undone some of its already executed actions. So, before executing an action, the agent should check all necessary preconditions hold. Note it should only *check*, thereby avoiding abducing again a plan for them: this way, if the preconditions hold the agent can continue and execute the planned action. The agent should only take measures to enforce the preconditions again whenever the check fails. Clearly, an $inspect$ion of the preconditions is what we need here.

6 Acknowledgements

We thank Robert Kowalski, Verónica Dahl and Henning Christiansen for discussions, Pierangelo Dell'Acqua for the declarative semantics, and Gonçalo Lopes for help with the XSB-XASP implementation.

References

1. J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, July 2004.
2. H. Christiansen and V. Dahl. Hyprolog: A new logic programming language with assumptions and abduction. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *LNCS*, pages 159–173. Springer, 2005.
3. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and advanced frontends (system description). In *12th Workshop on Logic Programming*, 1997.
4. M. Denecker and D. De Schreye. Sldnfa: An abductive procedure for normal abductive programs. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 686–700, Washington, USA, 1992. The MIT Press.
5. T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1–2):129–177, 1997.
6. T. H. Fung and R. Kowalski. The iff proof procedure for abductive logic programming. *J. Log. Prog.*, 33(2):151 – 165, 1997.
7. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of ACM*, 38(3):620–650, 1991.
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
9. K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996.

10. A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in AI and LP*, volume 5, pages 235–324. Oxford University Press, 1998.
11. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Procs. 4th Intl. Conf. Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, July 1997.
12. F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In E. Lamma and P. Mello, editors, *AI*IA*, volume 1792 of *LNCS*, pages 49–60. Springer, 1999.
13. T. Swift and D. S. Warren. An abstract machine for slg resolution: Definite programs. In *Symp. on Logic Programming*, pages 633–652, 1994.