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*) strictly within the abductive solution found, but without performing additional abductions, and without having to produce a complete model to do so. That is, such literals may consume, but not produce, the abduced literals in 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 how, by means of examples, one can employ it to investigate side-effects of interest (the *inspection points*) in order to help choose among abductive solutions, to express intentions, to implement deontic verifiers, to code strict preferences, as well as to permit use of passive integrity constraints – i.e. those not allowed to abduce in order to be satisfied but just to check for satisfaction without further abduction. The touchstone of Inspection Points can be construed as a form of meta-abduction, by meta-abducing an abduction to check (i.e. passively verify) that a certain concrete abduction is indeed adopted in an otherwise purported abductive solution. We furthermore show how to implement 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, Forward Chaining

1 Introduction

In this paper we present a new reasoning mechanism for abductive logic programs — the inspection points — which permits a selective forward chaining, from adopted abductions. This allows for a panoply of different tools such as deontic verifiers, strict preferences, and side-effects checking.

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 describe our implementation of the inspection points, illustrate it with an example, and compare it to other previous works in this direction.

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

1.1 Motivation

Sometimes, besides needing to abductively discover which actions to undertake in order to satisfy some condition, we may also want to know some of the side-effects of those actions; 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 actions, since some of them will be irrelevant to our concern. Similarly, the side-effects resulting from an abductive explanation might not all be of interest.

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

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

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 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.

1.2 Background Notation and Definitions

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

We call L 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.

Definition 2. ***Logic Program.*** *A Logic Program (LP for short) P is a (possibly infinite) set of ground Logic Rules.*

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

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

In this paper we focus solely on Normal LPs (NLPs), those whose heads of rules are positive literals, i.e., simple atoms; and there is default negation just in the bodies of the rules. Hence, when we just write "program" or "logic program" we mean a NLP.

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.

2 Reasoning with Logic Programs

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 conditions), 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, inspection points can be construed as utilizing a form of meta-abduction, by "meta-abducing" the specific abduction of actually checking (i.e. passively verify) that a certain and corresponding concrete abduction is indeed adopted. That is, one 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 abduced check. In our approach the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals 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. 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 intelligent solution, when one is focused on obtaining some specific conclusions of a set of premises, is afforded by selective 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 forward-chaining fashion. Combining backward-chaining with forward-chaining (and in particular with selective forward-chaining) would allow 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, just passively (though not destructively) consuming abducibles that are "produced" by abduction for the top query elsewhere.

In the sequel, we shall show how such a selective forward chaining from a set of hypotheses can actually be achieved 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* [16] or *DVL* [5] are adequate because they can indeed compute whole models for the program. We 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 [11]. 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 just 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 ([1–3, 6–8, 10, 12, 13]) 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 described in section 3. Though WFS is 3-valued, the abduction mechanism it employs can be, and in our case is, 2-valued. However, if one desires a 2-valued answer, a different semantics must be used. We suggest one such (relevant) semantics in subsection 2.5 — the Revised Stable Models [18].

The kind of situation we address with this contribution occurs when one wants only to passively determine which abductions, activated or produced by other goals, would be sufficient to satisfy some goal, but without actually abducing them, just consuming other goal's produced abductions. The difference between this and usual abductive query answering is subtle but of utmost importance, as it carries in itself the seed for a new primitive construct. This mechanism is that of *inspecting without abducing*, and it can be implemented by means of *meta-abduction*, discussed in detail in subsection 2.4.

2.4 Inspection Points

Meta-Abduction Meta-abduction is used in *abduction avoidant inspection*. Intuitively, it is 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-based system, and used to reify mechanisms as diverse as:

- Intentions — future goals which are not yet scheduled for solution, but that we know we want to have solved by abducing actions at some point in time.
- Deontic verifiers — a deontic verifier is simply a mechanism for checking or imposing some conditions. Typically, deontic verifiers are used to express what is allowed, what is forbidden, and what is obligatory, with application to moral reasoning.
- Side-effects — meta-abduction can be used to check if some literal is a consequence of the abductions made when solving the top goal.
- Strict preferences — the complex subject of preferences is outside the scope of this paper, but a short word on "strict preferences" and how meta-abduction can be used to implement them is provided in section 3.

Further explanation of these points — with illustrative examples — is given in section 3, right after the detailed outline of the implementation.

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 abductions is disallowed. In this setting, *inspection points* are the literals 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.

Declarative Semantics of Inspection Points The following simple transformation maps programs with inspection points into programs without them. Mark that the abductive stable models of the transformed program where each $abduced(x)$ is matched by the abducible x clearly correspond to the intended procedural meanings ascribed to the inspection points of the original program.

Definition 4. ***Transforming Inspection Points.*** *Let P 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 replacing:*
   - $inspect(not\ L)$ *with* $not\ inspect(L)$
   - $inspect(a)$ *or* $inspect(abduced(a))$ *with* $abduced(a)$ *if a is an abducible, and keeping* $inspect(L)$ *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* $1 \leq i \leq t$

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

Example 3. **Transforming a Program with Nested Inspection Levels.**
Let P be

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

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.

2.5 Alternative Semantics

The SM semantics does not guarantee existence of a model for every Normal Logic Program, and it does not enjoy relevancy either. Thus it cannot be used to provide an (abductive) answer to a query by need, in a top-down fashion. The lack of relevancy results because the SM semantics gives no semantics to *odd loops over negation — OLONs*[1]. In fact, OLONs are used as Integrity Constraints by the SMs programming community.

Example 4. **Odd Loop Over Negation as Integrity Constraint.**

$$a \leftarrow not\ a, X.$$

The Odd Loop Over Negation on literal 'a' prevents the remaining body X from being in any model.

Preferred Extensions In 1995, Dung proposed the preferred extensions [9] — a conservative extension to SMs which coincides with the SMs for all literals not involved in OLONs. What the Preferred Extensions do is assign all literals in the OLON (and their consequences) the *undefined* truth-value. All NLPs have a preferred extension, but if one wishes a total 2-valued semantics to every NLP the Preferred Extensions are not enough.

Revised Stable Models In 2005, Pereira and Pinto defined the Revised Stable Models (RSM) semantics [18], a 2-valued conservative extension to SM semantics guaranteeing existence of a model for every NLP, that enjoys relevancy and cumulativity. How the RSMs semantics handles OLONs is beyond the scope of this paper but the interested reader can find all the details in [18]. For our present matters, what is important is that the RSMs permits abductive query solving by need.

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 lays the foundations for efficiently computing queries over ground 3-valued abductive frameworks for extended logic programs with integrity constraints, on the WFS semantics and its partial SMs. 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

[1] An OLON is a cycle in the dependency call-graph where there is an odd number of default negations along the cycle from one literal back to itself.

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 [21] 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.

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. We assume such a "depth-first" discipline in the implementation of Inspection Points, described in detail below. Later we lift this restriction at the end of this 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, '$abduced(A)$' 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 '$abduced(X)$' 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 '$abduced(X)$') 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 ('$abduced(X)$' 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 '$abduced(X)$' 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 '$abduced(X)$' 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 simple 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 helps to clarify this.

Example 5. **Nested Inspection Points.** Consider the program from example 3, where the abducibles are a, b, c, d.

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 $abduced(b)$ 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.,
   $abduced(not\ d)$ 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 $abduced(c)$ 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: $[a, abduced(b), abduced(not\ d), abduced(c), b, c, not\ d]$;
13. the abductive solution is now checked for matches between meta-abductions and actual abductions. In this case, for every $abduced(A)$ in the abduction list there is an A also in the abduction list, i.e., every intention of abduction $abduced(A)$ 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 $abduced(A)$ and the corresponding A appear and were placed in the abductions list.

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

The actual 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.

3.2 Topical Application Examples

Intentions: When developing an agent's architecture one typically considers the *cycle of the agent* where perception, reasoning and action take place. The action part can consist simply of updating the agent's "to-do list" with another intended goal that will be

attended to when the best opportunity arises. The conditions for updating the intended goals can occur unexpectedly and they might be met as side-effects of other actions the agent takes. In such cases, the agent just wants to check if, by any chance, the conditions for some action become *true* while the agent undertakes the solution to some other problem. It does not need to make those conditions true by abducing actions to that effect, as it would deviate efforts from the problem it is undertaking.

Example 6. **Intentions.**

$$add\_new\_intention(X) \leftarrow inspect(ConditionForIntendingGoal X)$$

$inspect/1$ is what we use to adopt the meta-abduction mode.

Notice that this mechanism allows for explicitly taking advantage of computations already made. The fact that the rule above depends on an *inspect* means that there will be no extra abductions made in order to make $add\_new\_intention$ *true*; the $inspect$ will only check if, with the abductions made in the present derivation, even if not yet achieved, we can add the new intention.

Deontic Verifiers: The meta-abduction mechanism itself is implemented as we described: by means of abducing a special reserved unary predicate expressing the intention of actual abduction. *A posteriori*, after a (meta-)abductive answer to the query is found, different kinds of check/enforce restrictions can be put on the meta-abductions.

In our current implementation, in particular, we have just implemented the "meta-abduction $\Leftrightarrow$ abduction" enforcement check, which in practice corresponds to deontic verifiers: whenever we have $abduced(a)$ in the abductive answer we *must* also have a or else the whole answer will fail. Likewise, having $abduced(not\ a)$ in the abductive answer imposes that $not\ a$ has been also abduced, but not below an inspection of the triggering goal of $abduced/1$.

Of course, additional flexibility can be implemented via different reserved unary predicates for meta-abduction, each of which requiring a different level of "commitment" concerning meta-abductions. The $abduced/1$ predicate has now the meaning of *obligatory*. An extra possibility would be to have another predicate like $allowed/1$ which would, by itself, never prevent the abductive answer from succeeding, by not requiring its argument to be actually abduced, yet requiring its negation-complementary not to be explicitly abduced. This latter scenario allows for a numeric (frequentist) approach where an abductive answer can be rated according to the number of allowed actual abductions, i.e., $\frac{\#\{X:allowed(X)\wedge abduced(X)\}}{\#\{X:allowed(X)\}}$, or simply according to the number of allowed requests.

Example 7. **A gourmet's deontic principles.** For a gourmet, a proper meal goes along with red wine if the main dish is meat, and with white wine if it is fish. The abducibles are $merlot$, $chardonnay$, $meat$, and $fish$.

$$\begin{array}{ll} \leftarrow not\ proper\_meal & main\_dish \leftarrow meat \\ proper\_meal \leftarrow wine, main\_dish & main\_dish \leftarrow fish \\ wine \leftarrow red\_wine, inspect(meat) & red\_wine \leftarrow merlot \\ wine \leftarrow white\_wine, inspect(fish) & white\_wine \leftarrow chardonnay \end{array}$$

Specific wine abduction can be done before main dish abduction, without comitting to the kind of main dish. In fact, the abduction for the wine and the one for the main dish can be run in parallel. If the resulting combination of abductions is not acceptable according to the gourmet's deontic principles, an alternative abduction for the wine can be produced.

Side-Effects: The actual semantics of checking side-effects can be of two kinds:

1. We can require that some literal a is a side-effect consequence of the abductions made during the process of finding a solution for the query — this is the semantics we implemented.
2. We may want to know the truth-value (2 or 3 valued, depending on the specific application being developed) of some literal a as a consequence of the abductions made when finding an answer for the query. In this case, the reserved predicate $abduced$ would need to be binary — $abduced/2$ in order to provide the truth-value of the inspected literal as the second argument. The collected truth-values of the inspected literals can then be post-processed after the abductive answer is computed.

Strict Preferences: In an abductive framework, general preferences between abducibles can be expressed via a set of rules where abducibles are preferred only when they are "considered". An abducible is considered when it is abduced, but only if there is expectation in favor of it (since not all abducibles need be expected and thus made available in some situation), and also there is no expectation to the contrary, i.e., the abducible is not made available. Expectation for an abducible is expressed by application-specific conditions; whereas expectation against the abducible can be used to express the conditions under which an abducible is less preferred than another. These notions of preferences, consideration, expectation and contrary expectation are borrowed from [17]. An example will help clarify this.

Example 8. **Preferences.** Consider abducibles a and b. Under conditions c, a is more preferred than b — abstractly written as $a \lhd b \leftarrow c$. This means that whenever c holds and b is available, a must also be available.

The general rule for considering an abducible is
$consider(X) \leftarrow expect(X), not\ expect\_not(X), X$

The rules for saying that some abducible is expected are problem-domain specific and look like $expect(a) \leftarrow$ 'some conditions under which a is expectable'. This way, the rules for expressing preferences just need to undermine the *unpreferred* abducible whenever the conditions for the preference hold, i.e., consideration of an abducible is defeasible. This is done by $expect\_not$. The preference $a \lhd b \leftarrow c$ above is thus coded as $expect\_not(b) \leftarrow c, not\ consider(a)$

Since consideration of an abducible depends on its actual abduction, according to the rule for above, in order to have $expect\_not(b)$ true, a must not be considered. In case there is evidence in favor of expecting a and no evidence for expecting $not\ a$, this means a cannot be abduced. So, this kind of general preference may have an impact on the actual abduction (or not) of a.

Strict preferences, however, have a slightly different meaning. When we say we strictly prefer a to b, given conditions c, we mean that b can only be available if a is available, given that c holds. We write such a strict preference as $a \lll b \leftarrow c$, and this translates into $expect\_not(b) \leftarrow c, inspect(not\ consider(a))$, because with strict preferences we wish to forbid the preference to interfere with the actual abduction or not of a: this kind of preference has a mere verification role, whereas general preferences may condition which abductions are actually made in order to satisfy the preference. This is why strict preferences must recur to a mechanism like inspection points, just like in the rule $expect\_not(b) \leftarrow c, inspect(not\ consider(a))$ above.

Collaborative/Competitive Planning: An application of Inspection Points is Planning in a multi-agent setting. Whenever there are several agents there can be both collaboration and competition. Our agent may have a plan already set up and, in the course of carrying out the actions of the plan, it may find out that another agent has sabotaged his work by undoing some of the planned and already executed actions. In this case, before executing any action, our agent should check if all the necessary preconditions hold. Notice the agent should only *check*: 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 whenever the check fails, and only in those cases. Clearly, an $inspect$ion of the preconditions is what we need here.

On the other hand, the "other" agent can actually be collaborating with our agent; and when our agent goes on to execute some planned action, it might find that the companion agent has already done some of the work for him. In this case, our agent also wants only to check, not perform any actions in order to ensure the conditions. Merely checking if an intermediate goal has already been achieved — because some other agent helped, for example — is what we need to take advantage of this collaborative scenario. Again, an $inspect$ion is the way to implement such check.

Plausible reasoning: Creating arbitrary abductive theories to explain observations is easy; not so easy is to create minimal, consistent and plausible abductive theories. Plausibility is one of the strongest criteria when finding a rational and consistent explanation for observations. To impose plausibility on certain more unlikely abducibles, e.g., by means of an inspection point, is assuring the abducible (in case it is included in an abductive solution) is adopted only when there is a plausible reason to it. Example 2 illustrates a case of plausibility reasoning, where, by means of an *inspection point*, we enforce the abductive explanation for '$smoke$' to be '$tear\_gas$' only in case there is some action that caused it; in the example the action comes from the riot containment the police enforced.

3.3 Comparing with other systems

We briefly compared our abduction system with inspection points to HyProlog [4]. HyProlog 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. Moreover, HyProlog still has no declarative semantics, as opposed to our Inspection Points approach.

In [20], the authors detect a problem with the IFF abductive proof procedure [14] of Kung 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. The declarative semantics of our approach is supplied by a straight forward, meaning preserving, program transformation whose correctness is apparent.

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. Besides allowing for implementation of side-effects inspection, the meta-abduction principle, which is the heart of this contribution, permits a panoply of other reasoning tools such as deontic-verifiers (useful for moral reasoning [19]), and strict preferences. 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 the WFS with abduction. However, in case we need a total 2-valued semantics we need to recur to Revised Stable Models if we want to keep taking advantage of relevancy for top-down querying. Hence, our future work directions include extending the RSMs semantics with abduction, and, of course, meta-abduction. An efficient implementation of this semantics is also under way.

5 Acknowledgements

We thank Robert A. Kowalski and Verónica Dahl and Henning Christiansen for their insightful discussions and references to related works [4, 20].

References

1. J. Alferes, J. Leite, L. Pereira, and P. Quaresma. Planning as abductive updating. In D. Kitchin, editor, *Procs. of AISB'00*, 2000.
2. J. J. Alferes, L. M. Pereira, and T. Swift. Well-founded abduction via tabled dual programs. In *Intl. Conf. on Logic Programming*, pages 426–440, 1999.
3. J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *TPLP*, 4(4):383–428, July 2004.
4. 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.
5. 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 *Workshop Logische Programmierung*, 1997.
6. L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *J. of Logic and Computation*, 1(5):661–690, 1991.
7. M. Denecker and D. De Schreye. Sldnfa: An abductive procedure for normal abductive programs. In Apt, editor, *Procs. of the Joint Intl. Conf. and Symposium on Logic Programming*, pages 686–700, Washington, USA, 1992. The MIT Press.
8. P. M. Dung. Negations as hypotheses: An abductive foundation for logic programming. In *ICLP*, pages 3–17. MIT Press, 1991.
9. P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.
10. T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1–2):129–177, 1997.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
12. K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996.
13. A. C. Kakas and F. Riguzzi. Learning with abduction. In S. Džeroski and N. Lavrač, editors, *Procs. of the 7th Intl. W. on Inductive Logic Programming*, volume 1297, pages 181–188. Springer-Verlag, 1997.
14. R. Kowalski and T.H. Kung. The iff procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.
15. J. Neves, M. F. Santos, and J. Machado, editors. *Progress in Artificial Intelligence, 13th Portuguese Conference on Aritficial Intelligence, EPIA 2007, Workshops: GAIW, AIASTS, ALEA, AMITA, BAOSW, BI, CMBSB, IROBOT, MASTA, STCS, and TEMA, Guimarães, Portugal, December 3-7, 2007, Proceedings*, volume 4874 of *LNCS*. Springer, 2007.
16. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Procs. of the 4th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, July 1997.
17. L. M. Pereira and G. Lopes. Prospective logic agents. In Neves et al. [15], pages 73–86.
18. L. M. Pereira and A. M. Pinto. Revised stable models - a semantics for logic programs. In G. Dias et al., editor, *Progress in AI*, volume 3808 of *LNCS*, pages 29–42. Springer, 2005.
19. L. M. Pereira and A. Saptawijaya. Modelling morality with prospective logic. In Neves et al. [15], pages 99–111.
20. 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.
21. T. Swift and D. S. Warren. An abstract machine for slg resolution: Definite programs. In *SLP*, pages 633–652, 1994.