Artificial Intelligence Review (2024) 57:334
https://doi.org/10.1007/s10462-024-10928-7

An efficient propositional system for Abductive Logic Programming

Marco Gavanelli[1] · **Pascual Julián-Iranzo**[2] · **Fernando Sáenz-Pérez**[3]

Accepted: 22 August 2024

Abstract
Abductive logic programming (ALP) extends logic programming with hypothetical reasoning by means of abducibles, an extension able to handle interesting problems, such as diagnosis, planning, and verification with formal methods. Implementations of this extension have been using Prolog meta-interpreters and Prolog programs with Constraint Handling Rules (CHR). While the latter adds a clean and efficient interface to the host system, it still suffers in performance for large programs. Here, the concern is to obtain a more performant implementation of the SCIFF system following a compiled approach. This paper, as a first step in this long term goal, sets out a propositional ALP system following SCIFF, eliminating the need for CHR and achieving better performance.

Keywords Abductive Logic Programming · SCIFF · Hypothetical Reasoning · System Implementation

1 Introduction

Historically, since Charles Sanders Peirce (Peirce 1965), abduction[1] was seen as an explanatory reasoning activity for generating hypotheses. It is the case of experimental sciences when the scientific method is used.

[1] Which he also called "making a hypothesis".

Marco Gavanelli, Pascual Julián-Iranzo and Fernando Sáenz-Pérez contributed equally to this work.

✉ Marco Gavanelli
marco.gavanelli@unife.it

Pascual Julián-Iranzo
Pascual.Julian@uclm.es

Fernando Sáenz-Pérez
fernan@ucm.es

[1] Department of Engineering, Ferrara University, Ferrara, Italy

[2] Dept. of Information Technologies and Systems, University of Castilla-La Mancha, Ciudad Real, Spain

[3] Dept. of Software Engineering and Artificial Intelligence, Universidad Complutense de Madrid, Madrid, Spain

Published online: 17 October 2024

Abduction is one of the three major types of inference, jointly with deduction and induction. Deduction infers a conclusion from the premises by applying some inference rules, and when the premises are true, the conclusion must necessarily be true. However, conclusions in induction and abduction inferences are not necessarily true. To illustrate this last point and the mediate inference nature of abduction, (1965, CP 2.623) poses the following scenario:

> Suppose I enter a room and there find a number of bags, containing different kinds of beans. On the table there is a handful of white beans; and, after some searching, I find one of the bags contains white beans only. I at once infer as a probability, or as a fair guess, that this handful was taken out of that bag.

and compares abduction with the other types of inference:

DEDUCTION	
Rule:	All the beans from this bag are white.
Case:	These beans are from this bag.
Result:	These beans are [certainly] white.
INDUCTION	
Case:	These beans are from this bag.
Result:	These beans are white.
Rule:	All the beans from this bag are [probably] white.
ABDUCTION	
Rule:	All the beans from this bag are white.
Result:	These beans are white.
Case:	These beans are [possibly] from this bag.

The tags "certainly", "probably", and "possibly" are added to clarify the necessary/non-necessary quality of each inference. Note that induction infers some general principle of a quantity of observed statistical information, but it does not assure the truth of that general principle. For example, someone who has always lived in Kenya and has not watched BBC wildlife documentaries might infer "All elephants have big ears" from the observation that 100% of the elephants he saw have big ears. On the other hand, induction is synthetic or ampliative, meaning that the conclusion of an induction goes beyond what is (logically) contained in the premises, which is why it is a non-necessary inference but probable (that is, the conclusion is held to be true on insufficient grounds).

For Peirce, abduction is also a form of synthetic reasoning but, starting from a (general) rule and a result, it obtains a case (an abducible). The product of abductive reasoning is an abducible hypothesis that, if assumed, would explain the result. Consequently, Peirce conceives abduction as a mediate inference of a synthetic, probable and explanatory nature. Although both abduction and induction are ampliative, in abduction there is an implicit or explicit appeal to explanatory considerations, whereas in induction there is only an appeal to observed frequencies or statistics. Also, what their conclusions offer differs in terms of their ampliative character. Thus, induction infers from a set of facts another set of similar facts, so its enlargement is quantitative. On the other hand, abduction infers from facts of one kind, facts of a different kind, which is why Peirce sometimes calls its enlargement "qualitative".

Now abduction mostly refers to the activity of explanatory reasoning in justifying hypotheses. Then, abduction is also often called "Inference to the Best Explanation". In this last sense, abduction can be defined as (Douven 2021):

> Given evidence E and candidate explanations $H_1, \ldots, H_n$ of E, if H_i explains E better than any of the other hypotheses, infer that H_i is closer to the truth than any of the other hypotheses.

Note that this definition requires an account of closeness to the truth. How to establish a criterion for selecting the best hypothesis is an active research field. For instance, in Abductive Logic Programming, Caroprese et al. (2014, 2022) introduced a measure of the simplicity of an explanation based on its degree of arbitrariness, with the less arbitrary explanations being the preferred ones.

This study has more humble goals, aiming to obtain a set of those abducible hypotheses that allow the observed evidence to be supported. Informally, an abductive inference process will be used which will select an abducible hypothesis (from a set), if it is necessary in order to be able to infer the evidence in the context of a theory. This process will be able to obtain all the necessary abducibles independently of their degree of support for the evidence. For instance, assume this simple scenario "Bob is known to ski on Saturdays if it is not snowing. And when he is not skiing, he is on campus" and the observation that "Bob is on campus". The system to be described will obtain, as an explanation of this fact, the abducibles that it is not Saturday or it is snowing, because one of these abducibles is necessary to infer that Bob is on campus, but no preference will be given to one answer over the other.

Abductive logic programming (ALP) extends logic programming with hypothetical reasoning by means of abducibles, which provide a form of explanation to goals, and integrity constraints (ICs). ALP operational semantics is usually given as a proof procedure (Kakas et al. 1992), and was first implemented with Prolog metainterpreters (Fung and Kowalski 1997). Given the relationship between ALP and constraints, another implementation method was developed by using Constraint Handling Rules (CHR (Frühwirth 1998)), either translating ICs directly into CHR rules (Abdennadher and Christiansen 2001; Gavanelli et al. 2003; Christiansen and Dahl 2005) or mapping abducibles onto CHR constraints and implementing transitions as CHR rules (Alberti et al. 2013). Thus, in this last system, the implementation closely follows operational semantics transitions. In addition, the tight integration of CHR and the host language Prolog in which this approach is implemented allows underlying libraries to be used and pure Prolog code to be executed with no slow-downs.

Despite all these advantages and the effectiveness of CHR implementations, they still suffer from a performance burden due to handling the constraint store, and propagation. Furthermore, abducibles that are assumed during solving (which are represented as CHR constraints) must be compiled via `term_expansion/2`, and dynamic type checking is applied to arguments at runtime, which amounts to an extra burden. When dealing with large programs (such as in bi-abduction for reasoning techniques about memory properties with millions of LOCs (Calcagno et al. 2011)), using CHR can be a shortcoming due to time performance; other, more effective alternatives might therefore be devised. This study explores a compiled approach to transforming ALP programs into Prolog programs, including the management of ICs. As ICs are thought to prune the computation path in advance, they should be triggered as soon as possible, much like a CHR implementation would , but avoiding the performance burdens of such a constraint propagation approach.

The long-term goal is to obtain a more performant implementation of the SCIFF system (Alberti et al. 2013). Thus, as a first step towards this goal, this study sets out PIFFC (Propositional IFF Compiler) as an efficient alternative for dealing with the propositional version of ALP, which will be mainly compared with the SCIFF state-of-the-art system. Following (Julián-Iranzo and Sáenz-Pérez 2021), a compiled approach with the use of the dynamic database for assumed abducibles is developed for the ALP case as found in SCIFF.

2 Background

According to Fung and Kowalski (1997), an abductive logic program $\Pi = \langle KB, IC, Ab \rangle$ is composed of a set of rules KB (i.e., the knowledge base of Π or theory) and integrity constraints IC (i.e., restrictions that must be satisfied in Π) over both defined ($\mathfrak{D}$) and (Ab) abducible predicates. A defined predicate $p \in \mathfrak{D}$ consists, at least, of a rule for p in KB, whereas an abducible predicate $a \in Ab$ is undefined but can be assumed to be true throughout a proof.

Each rule in KB has the form:

$$h \leftarrow l_1 \wedge \ldots \wedge l_n.$$

where h, called the *head*, is a propositional (user-)defined predicate (an atom), and $l_1 \wedge \ldots \wedge l_n$, called the *body*, is a conjunction of literals. A literal l_i is either an atom p or negated atom $\neg p$ (written as `naf(p)` in the concrete syntax as per (Alberti et al. 2022)), where p can be either a defined or an abducible predicate. Each abducible a in Ab is tagged in the concrete syntax as `abd(a)`. An empty conjunction (in the body) is read as *true*.

An integrity constraint (IC) represents a logical implication *Left* $\rightarrow$ *Right*. Here, the syntax of the actual implementation of the SCIFF system (Alberti et al. 2013, 2022):

$$l_1 \wedge \ldots \wedge l_n \rightarrow (l_{11} \wedge \ldots \wedge l_{1n_1}) \vee \ldots \vee (l_{k1} \wedge \ldots \wedge l_{kn_k})$$

where the l's are literals, is also followed. Similarly to rules, sometimes *Left* is called the *body* and *Right* the *head*. An empty disjunction is read as *false*.

The semantics of this language conforms to the one established by Fung and Kowalski (1997) where an answer to a goal G is the set of abducibles $\Delta \subseteq Ab$ such that the following entailments (under the 3-valued completion semantics (Kunen 1987)) hold:

$$KB \cup \Delta \models IC$$
$$KB \cup \Delta \models G.$$

Operationally, a semantics is here proposed that follows Prolog SLDNF, extended with the notions of abducibles and integrity constraints. ICs in a program can be understood as restrictions on possible answers to a goal under SLDNF. In an implication *Left* $\rightarrow$ *Right*, *Left* is only checked (there is no attempt to prove it; e.g., an abducible in *Left* is only checked, not added). If *Left* holds, *Right* contains generic goals (such as those found in the right-hand side of user rules, including abducibles that can be assumed) that must hold to make the implication true.

Goal solving proceeds by following top-down, left-to-right SLDNF resolution. Defined predicates are solved by this procedure. However, each time a call to an abducible succeeds, it is assumed as true and added to Δ (which is initially empty) so that further calls to the abducible will simply succeed, and the proof continues under this assumption. In addition, the set of ICs must be checked.

One possible approach to checking ICs is at the end of goal solving, for each different alternative (based on backtracking) of the top-level user goal. However, in order to prune computations in advance, after any call ϕ, each IC with ϕ in its condition can be triggered. Thus:

- Each time a goal p is solved, each IC (not tested yet) with p in its condition is triggered, omitting the call to p itself in the condition, since it would obviously succeed. The goal p can be either a defined or an abducible predicate.

- When testing an IC condition c, no abducibles are assumed, but they are simply checked (Fung and Kowalski 1997). This means that other ICs attached to any defined predicate in the proof tree of c will be triggered, but ICs attached to abducible predicates should not be, because they were triggered for the previous assumptions.
- For a goal $\neg p$, rewriting to the equivalent formula $p \rightarrow false$ is used, as in Fung and Kowalski (1997), Alberti et al. (2022)). Once $\neg p$ is reached in a proof, the IC $p \rightarrow false$ is checked and also added to the current set of ICs. Thus, while KB is constant when solving a goal, IC is not because it may increase due to negations. Details are given in Sects. 3 and 4, also explaining a rewriting to avoid negations in IC conditions.
- Once the conclusion of an IC succeeds, there is no need to recheck it because the truth values of its predicates cannot change. In particular, if a negated abducible p succeeds, the IC $p \rightarrow false$ is added as explained in the previous item, therefore preventing p from becoming true in following computations. ICs whose conclusion has succeeded are called *checked constraints*. After goal solving, any IC that is not checked must be triggered.

The next section describes an operational semantics that takes into account these considerations; in particular, keeping track of already checked ICs.

3 Operational semantics

This section describes the operational semantics to be implemented in Sect. 4. This intended semantics is defined by a *state transition system*, which is made up of a notion of state $\langle IC, \Delta, N, I, \phi \rangle$ and a state transition relation $\Rightarrow$, where: IC is a set of implications, initially equal to the set of all the integrity constraints stated by the user; $\Delta \subseteq Ab$ includes all the abducible predicates that have been assumed; N is the sequence of ICs in IC being checked, which we call *checking constraints*; elements of N are separated by dots (an empty sequence is written as ϵ and, by abuse of notation, set operations are used on sequences, such as: $I \cup N \equiv I \cup \bigcup\{i_i\}$, with $N = i_1 \ldots i_n$); $I \subseteq IC$ includes all the ICs that have been checked (i.e., those with both a condition and a conclusion that are satisfied);[2] and ϕ is a goal. Checking ICs are introduced in the state to prevent (infinitely) rechecking an IC that is being checked. As will become clear, all transitions handle the N sequence as a stack of implications, and the following invariant is always true: each time the current goal ϕ is an implication $A \rightarrow B$, the stack is of the form $N.i$ and the top of the stack, i represents the implication currently being propagated, which was (possibly) rewritten into $A \rightarrow B$.

In a sequence of transition steps, the initial state is $\langle IC, \emptyset, \epsilon, \emptyset, \phi \rangle$ and the final state is either $\langle IC', \Delta, \epsilon, I, true \rangle$ (a *successful* state) or $\langle IC', \Delta, N, I, false \rangle$ (a *failure* state is reached when no transition step can be applied). Sometimes, *derivation* is used to refer to a sequence of transition steps and *proof* to refer to a successful sequence of transition steps.

Transition rules specify the relation $\Rightarrow$ in this state transition system, where the symbols l and p are respectively used for a literal and a predicate (either defined or abducible), and a when referring in particular to an abducible predicate; the functions $defs(L)$ and $abds(L)$ respectively return the set of defined and abducible literals in L. As usual, $\Rightarrow^*$ denotes zero or more steps of $\Rightarrow$. In each rule, the state below the line (the new state) can be inferred from

[2] Note that $IC \neq N \cup I$ in general because new constraints different from the original can be added to IC, stemming from the way negation is handled (see later), and furthermore a constraint in N can be suspended when it is being checked and therefore it must be deleted from N but not inserted in I. This justifies the need for the component IC in the notion of state.

the state above the line (the current state), in a single transition step, if the condition of the transition rule holds.

To formalize the transition rules it is first necessary to introduce the concept of satisfiability of integrity constraints. The *Sat* function takes the set IC of integrity constraints to be verified and the initial context IC_0, Δ_0, N_0, I_0, and returns either the output (IC_k, Δ_k, I_k) or $failure$.

Definition 3.1 Let IC be a set of non-checked, non-checking integrity constraints. IC is *satisfiable* starting from the set of integrity constraints IC_0, the set of abducibles Δ_0, checking ICs N_0 and checked ICs I_0, if given a set of indexes $k \in 1 \ldots m$, with $m = |IC|$, $\forall i_k \in IC$: $\exists \mathscr{D}_k \equiv \langle IC_{k-1}, \Delta_{k-1}, N_0.i_k, I_{k-1}, i_k \rangle \Rightarrow^* \langle IC_k, \Delta_k, N_0, I_k, true \rangle$. In such a case we write $Sat(IC, IC_0, \Delta_0, N_0, I_0) = (IC_m, \Delta_m, I_m)$. Otherwise, we write $Sat(IC, IC_0, \Delta_0, N_0, I_0) = failure$.

When it is required that there exist a proof for $\mathscr{D}_k$ in this definition for *Sat*, it may be the case that several proofs exist, possibly each giving an alternative answer. Sometimes, when a set of non checked integrity constraints IC is satisfiable but the result (IC_k, Δ_k, I_k) of the verification process is of no interest, $Sat(IC, \Delta_0, N_0, I_0) \neq failure$ is simply written. Finally, note that over the transition sequences it is possible to verify new integrity constraints due to the application of transition steps.

Transition rules are specified next:

- *Abduction (Abd).*

This rule is intended to handle assumptions of abducibles. As noted in Sect. 2, abducibles are not assumed when checking IC conditions. To disallow such assumptions, the witness element $\mathcal{B}$ is used in Δ to represent that assumptions are blocked. In any case, when an abducible a is to be assumed, any integrity constraint that has not been checked yet, and with a in its condition, should be tested. This is accomplished by the function *Sat*, which returns the new components of the state (new IC, Δ, and I sets). Recall that the set of integrity constraints may be enlarged because of negated calls as explained in the previous paragraphs (IC'' in this rule). When the abducible can be assumed, Δ is enlarged with the result of *Sat* and the abducible itself. If assumptions are not blocked and the abducible a is in the current Δ, this set does not change. In both cases, the set of integrity constraints that have been completely checked is also enlarged with the result of *Sat* (I').

1. If $\mathcal{B} \notin \Delta$ and $Sat(IC', IC, \Delta \cup \{a\}, N, I) = (IC'', \Delta', I')$, where $IC' = \{(L \rightarrow R) \in IC \backslash (I \cup N) \mid a \in abds(L)\}$:

$$\frac{\langle IC, \Delta, N, I, a \wedge l_1 \wedge \ldots \wedge l_n \rangle}{\langle IC \cup IC'', \Delta' \cup \Delta \cup \{a\}, N, I' \cup I, l_1 \wedge \ldots \wedge l_n \rangle}$$

2. If $\{\mathcal{B}, a\} \subseteq \Delta$ and $Sat(IC', IC, \Delta, N, I) = (IC'', \Delta, I')$, where $IC' = \{(L \rightarrow R) \in IC \backslash (I \cup N) \mid a \in abds(L)\}$:

$$\frac{\langle IC, \Delta, N, I, a \wedge l_1 \wedge \ldots \wedge l_n \rangle}{\langle IC \cup IC'', \Delta, N, I' \cup I, l_1 \wedge \ldots \wedge l_n \rangle}$$

- *Clause (Cla).*

This is the typical transition rule used to unfold a defined predicate by its definition. In this case, ICs containing p in their conditions and that are neither checked nor under checking should be checked, a procedure performed by the function *Sat*. As before, this function may return an enlarged set of ICs, abducibles and checked constraints.

If $p \leftarrow \phi \in KB$ and $Sat(IC'', IC, \Delta, N, I) = (IC', \Delta', I')$, where $IC'' = \{(L \rightarrow R) \in IC \backslash (I \cup N) \mid p \in defs(L)\}$:

$$\frac{\langle IC, \Delta, N, I, p \wedge l_1 \wedge \ldots \wedge l_n \rangle}{\langle IC \cup IC', \Delta' \cup \Delta, N, I' \cup I, \phi \wedge l_1 \wedge \ldots \wedge l_n \rangle}$$

Example 1 Consider the abductive program whose knowledge base $KB = \{p \leftarrow a \wedge q.\ q.\}$, the set of integrity constraints $IC = \{p \rightarrow b.\}$ and a and b are abducible. If the goal is p, it is substituted by $a \wedge q$ in the following state; also, the satisfiability of the integrity constraint $p \rightarrow b$ is tested by function *Sat*.

- *Negation (Naf).*
  In accordance with (Fung and Kowalski 1997; Alberti et al. 2022), negative literals $\neg p$ in a goal are rewritten into the equivalent form $p \rightarrow false$. In a second step, if p is a defined predicate, the atom p is unfolded with its definitions, recursively. This produces an unfolding tree whose leaves are $\mathscr{Q}_i$ and which leads to a set of constraints $\mathscr{Q}_i \rightarrow false$. It is important to note that the generation of the unfolding tree is guided by the dependency graph of the program, which helps to detect the halt condition when a negated literal is detected in a node of the tree. Since the unfolding steps might introduce negative literals in the condition of the implication, in a third step, each $\neg p_i$ in an IC condition $c_1 \wedge \ldots \wedge c_n \wedge \neg p_1 \wedge \ldots \wedge \neg p_m \rightarrow d_1 \vee \ldots \vee d_k$ is rewritten as a disjunction in the conclusion as: $c_1 \wedge \ldots \wedge c_n \rightarrow d_1 \vee \ldots \vee d_k \vee p_1 \vee \ldots \vee p_m$. These three steps are performed by the function *transform*.

Example 2 Let $KB = \{p \leftarrow q \wedge \neg r.\quad p \leftarrow a.\quad q \leftarrow b.\}$ be the knowledge base of an abductive program where a and b are abducible, and suppose the current goal is $\neg p$; it is first rewritten as $p \rightarrow false$, then substituted for its definitions

$$q \wedge \neg r \rightarrow false$$
$$a \rightarrow false$$

and recursively for the defined predicate q:

$$b \wedge \neg r \rightarrow false$$
$$a \rightarrow false$$

Then the negative literal $\neg r$ in the body is rewritten as follows

$$b \rightarrow r$$
$$a \rightarrow false.$$

Thus, if a literal $\neg p$ is in the current goal, it is translated into a set of integrity constraints $IC'' = transform(\neg p)$; if such integrity constraints are satisfiable, in the next state the literal $\neg p$ is substituted by *true* (or simply omitted if there are more conjuncts), otherwise the goal in the following state is *false*.

If $Sat(IC'', IC \cup IC'', \Delta, N, I) = (IC', \Delta', I')$, where $IC'' = transform(\neg p)$:

$$\frac{\langle IC, \Delta, N, I, \neg p \wedge l_1 \wedge \ldots \wedge l_n \rangle}{\langle IC', \Delta', N, I \cup I', l_1 \wedge \ldots \wedge l_n \rangle}$$

Otherwise:

$$\frac{\langle IC, \Delta, N, I, \neg p \wedge l_1 \wedge \ldots \wedge l_n \rangle}{\langle IC, \Delta, N, I, false \rangle}$$

It is important to note that in the test $Sat(IC'', IC \cup IC'', \Delta, N, I) = (IC', \Delta', I')$, the second argument must be $IC \cup IC''$ and not simply IC as one might naïvely think. E.g., assume that $IC'' = \{a \rightarrow false\}$ (where $a \rightarrow false$ is a new integrity constraint previously obtained by *transform*) and IC, Δ, N and I are empty. Then, by applying Definition 3.1, Sat returns $(\emptyset, \emptyset, \emptyset)$ and the new integrity constraint is lost. In order to ensure that the integrity constraint $a \rightarrow false$ is kept, it is necessary to pass the set $IC \cup IC''$ to the second argument of Sat. Thus: $Sat(\{a \rightarrow false\}, \{a \rightarrow false\}, \emptyset, \epsilon, \emptyset) = (\{a \rightarrow false\}, \emptyset, \emptyset)$.

- *Implication (Imp).*

This rule deals with implications of the form *Left* $\rightarrow$ *Right*, which represent integrity constraints. Recall that the witness element $\mathcal{B}$ is used to block possible assumptions in IC conditions. This element must be introduced in Δ if a conjunct in a condition is to be proved, and removed when the condition has been proven (either to *true* or *false*).

In the implications occurring in the initial state of each case identified below, i represents the current constraint under checking. In this initial state, in general i's condition has been proved up to a given conjunct c, i.e., the complete condition can contain other conjuncts to the left of c which have already been proved.

1. If $a \in \Delta$:

$$\frac{\langle IC, \Delta, N, I, a \wedge l_1 \wedge \ldots \wedge l_n \rightarrow \phi \rangle}{\langle IC, \Delta \cup \{\mathcal{B}\}, N, I, l_1 \wedge \ldots \wedge l_n \rightarrow \phi \rangle}$$

   In this first case, the abducible is already assumed and the derivation proceeds by adding the witness (should it not already be in Δ) and removing the abducible from the conjuncts. Otherwise ($a \notin \Delta$):

$$\frac{\langle IC, \Delta, N.i, I, a \wedge l_1 \wedge \ldots \wedge l_n \rightarrow \phi \rangle}{\langle IC, \Delta \setminus \{\mathcal{B}\}, N, I, true \rangle}$$

   Since the abducible a is not in Δ, the proof of i's condition stops; so, i is removed from N. The witness element $\mathcal{B}$ must be removed as well, indicating that the evaluation of the condition i has been temporarily suspended.
2. If $\exists \mathscr{D} \equiv (\langle IC, \Delta \cup \{\mathcal{B}\}, N, I, p \rangle \Rightarrow^* \langle IC', \Delta', N, I', true \rangle)$:

$$\frac{\langle IC, \Delta, N, I, p \wedge l_1 \wedge \ldots \wedge l_n \rightarrow \phi \rangle}{\langle IC', \Delta, N, I', l_1 \wedge \ldots \wedge l_n \rightarrow \phi \rangle}$$

   This rule corresponds to proving a defined predicate p occurring in the condition of an implication. A derivation is started for the goal p and, if such derivation succeeds, p can be removed from the condition. However, in the derivation for p, new abducibles should not be assumed, so the witness element $\mathcal{B}$ is added to the initial Δ. As assumptions are blocked, the resulting Δ' is $\Delta \cup \{\mathcal{B}\}$, and $\mathcal{B}$ is not needed in the final state since $\mathcal{B}$ is correspondingly added in each case of the transition rules when needed.
3. If $(\forall \mathscr{D})\mathscr{D} \equiv (\langle IC, \Delta \cup \{\mathcal{B}\}, N.i, I, p \rangle \Rightarrow^* \langle IC', \Delta', N', I', false \rangle)$:

$$\frac{\langle IC, \Delta, N.i, I, p \wedge l_1 \wedge \ldots \wedge l_n \rightarrow \phi \rangle}{\langle IC, \Delta \setminus \{\mathcal{B}\}, N, I, true \rangle}$$

   This rule corresponds to failing to derive a defined predicate p. As in Case 2 of this rule, assumptions are blocked and $\Delta' = \Delta \cup \{\mathcal{B}\}$. Since the evaluation of the implication is terminated, the witness $\mathcal{B}$ is removed from the final set of abduced literals, in case Δ contains it.

Example 3 Consider the implication $p \wedge r \rightarrow c$ together with the knowledge base $KB = \{p \leftarrow a \wedge q.\ \ q \leftarrow b.\}$ where a, b and c are abducible. If the set $\Delta = \{a\}$, a derivation is started with $\Delta' = \{a, \mathcal{B}\}$; in order to derive p, the abducible b must be true, but it cannot be assumed, since $\mathcal{B} \in \Delta'$; the derivation fails and the implication is substituted with *true*, as its body is *false*.
If, instead, $\Delta = \{a, b\}$, then the derivation succeeds, and the implication $p \wedge r \rightarrow c$ is rewritten as $r \rightarrow c$.

4. [No condition]

$$\frac{\langle IC, \Delta, N.i, I, true \rightarrow \phi\rangle}{\langle IC, \Delta \setminus \{\mathcal{B}\}, N, I \cup \{i\}, \phi\rangle}$$

In this case, a *true* condition means that the body holds (it was completely checked) and further transition steps must start without $\mathcal{B}$ in Δ. In the same way, i is removed from the sequence of checking constraints because its condition has already been proved to be true. Finally, i is added to the set of checked constraints because i will be completely checked when ϕ is satisfied by further derivation steps.

Note that if the initial state comes from the evaluation of a previous condition corresponding to another IC, that former condition will be responsible for adding again the witness when appropriate.

Example 4 Consider $i_1 \equiv p \wedge q \rightarrow Right$ and $i_2 \equiv p \rightarrow r$ under a program $KB = \{(p \leftarrow true), (r \leftarrow true)\}$. During the application of Sat to i_1, (Imp.2) is applied, looking for the satisfiability of i_2, with $\mathcal{B}$ being added. After applying (Cla) and obtaining *true* for the condition, $\mathcal{B}$ is removed from Δ. However, by continuing the derivation of i_1 condition, (Imp.2) is applied, which indeed adds $\mathcal{B}$ to Δ.

- *Disjunction (Dis).*

This rule simply creates alternative states from a disjunction of goals.

$$\frac{\langle IC, \Delta, N, I, \phi_1 \vee \phi_2\rangle}{\langle IC, \Delta, N, I, \phi_1\rangle} \qquad \frac{\langle IC, \Delta, N, I, \phi_1 \vee \phi_2\rangle}{\langle IC, \Delta, N, I, \phi_2\rangle}$$

Naturally, and as usual in any logic programming framework, if for a certain state $\langle IC, \Delta, N, I, \phi\rangle$ it is impossible to apply any of the above specified state transition rules, a failure state $\langle IC, \Delta, N, I, false\rangle$ is obtained.

Definition 3.2 Given an abductive logic program $\Pi = \langle KB, IC, Ab\rangle$, a *computed answer* for a user goal ϕ is any set of abducibles Δ' obtained after a successful sequence of transition steps:

$$\langle IC, \emptyset, \epsilon, \emptyset, \phi\rangle \Rightarrow^* \langle IC', \Delta, \epsilon, I, true\rangle$$

if $Sat(IC' \setminus I, IC', \Delta, \epsilon, I) = (IC'', \Delta', I')$.

In this condition, the function Sat checks that every integrity constraint in $IC' \setminus I$ (that has not been checked during the proof) holds. Also, in addition to Δ, new abducibles can be assumed, and they are delivered in the answer Δ'.

Example 5 Consider the adaptation to the PIFFC syntax of the abductive logic program that appears in (Fung and Kowalski 1997):

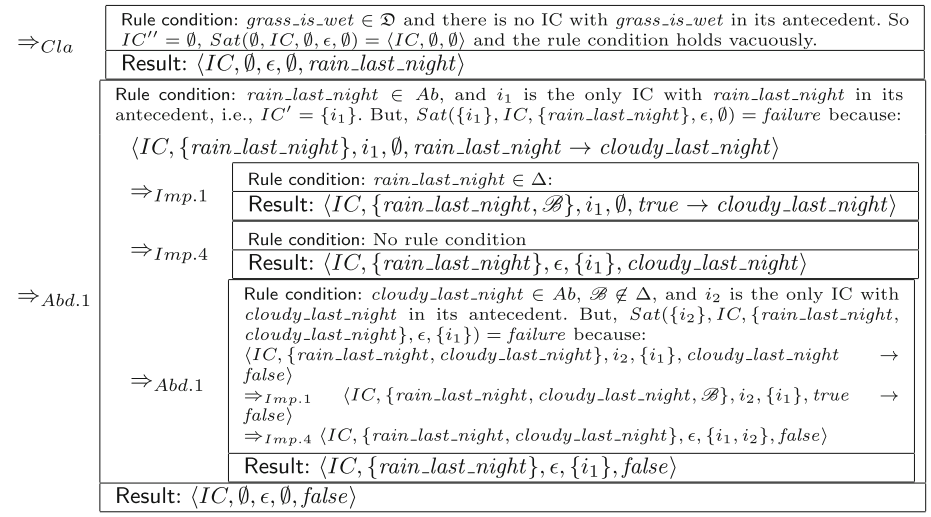

Fig. 1 A failure sequence of transitions steps for Example 5

KB r_1: *grass_is_wet* ← *rain_last_night*
r_2: *grass_is_wet* ← *sprinkler_was_on*
IC i_1: *rain_last_night* → *cloudy_last_night*
i_2: *cloudy_last_night* → *false*
Ab *rain_last_night*, *sprinkler_was_on*, *cloudy_last_night*
G *grass_is_wet*

For this program, the initial state is $\langle IC, \emptyset, \epsilon, \emptyset, grass\_is\_wet\rangle$ and there are only two possible transition sequences. The first sequence starts by applying a step with the (*Cla*)-rule for r_1. This is possible because $grass\_is\_wet$ belongs to $\mathfrak{D}$ and there does not exist a non-checked IC, $i_k \equiv (L \rightarrow R)$, such that $grass\_is\_wet \in defs(L)$. The whole sequence is shown in Fig. 1, where boxes detail auxiliary proofs, which are triggered in order to verify the applicability conditions of a transition rule. If the rule conditions hold, the next transition state is generated; otherwise, if no other transition rule is applicable, a failure state is produced. It can be seen that the (*Abd.1*)-rule cannot be applied because $Sat(\{i_1\}, IC, \{rain\_last\_night\}, \epsilon, \emptyset) = failure$, after a series of auxiliary derivations: $\langle IC, \{rain\_last\_night\}, i_1, \emptyset, rain\_last\_night \rightarrow cloudy\_last\_night\rangle \Rightarrow^* \langle IC, \{rain\_last\_night\}, \epsilon, \{i_1\}, false\rangle$ can be inferred by applying (*Imp.1*), (*Imp.4*) and (*Abd.1*). A failing rule condition occurs, preventing the (*Abd.1*) transition rule from being applied. Furthermore, no other transition rules are applicable (in particular, *Abd.2*); so, finally, after a cascade of failing rule conditions, the failure state $\langle IC, \emptyset, \epsilon, \emptyset, false\rangle$ is obtained. Thus, this branch leads to no answer.

Alternatively, the proof can be started with the (*Cla*)-rule for r_2, leading to $\langle IC, \emptyset, \epsilon, \emptyset, sprinkler\_was\_on\rangle$. Since the goal is an abducible predicate, the inference rule (*Abd.1*) is applied, for which there is no IC related to *sprinkler_was_on*; thus: $\langle IC, \{sprinkler\_was\_on\}, \epsilon, \emptyset, true\rangle$ is obtained. According to Definition 3.2, the abducible set in this tuple becomes (the only) answer provided that $Sat(\{i_1, i_2\}, \{sprinkler\_was\_on\}, \epsilon, \emptyset) \neq failure$:

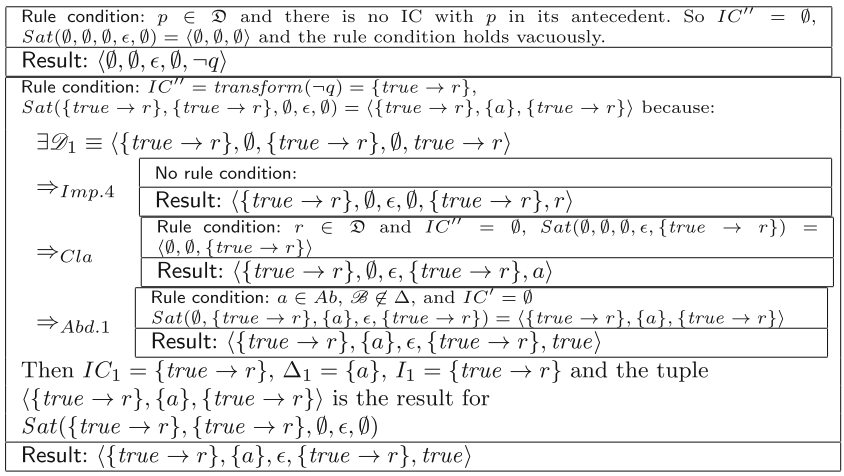

Fig. 2 A successful sequence of transition steps for Example 6

- first (for i_1):
  $\langle IC, \{sprinkler\_was\_on\}, i_1, \emptyset, rain\_last\_night \rightarrow cloudy\_last\_night\rangle \Rightarrow_{Imp.1}$
  $\langle IC, \{sprinkler\_was\_on\}, \epsilon, \emptyset, true\rangle$ and,
- second (for i_2):
  $\langle IC, \{sprinkler\_was\_on\}, i_2, \emptyset, cloudy\_last\_night \rightarrow false\rangle \Rightarrow_{Imp.1}$
  $\langle IC, \{sprinkler\_was\_on\}, \epsilon, \emptyset, true\rangle$.

□

Example 6 The following program serves to illustrate a transition sequence that includes the application of the (*Naf*)-rule. It is also interesting because, first, it shows a situation where new ICs are added to the initial set of ICs, and second because it includes alternative answers due to the *Sat* function:

KB $r_1: p \leftarrow \neg q$
$r_2: q \leftarrow \neg r$
$r_3: r \leftarrow a$
$r_2: r \leftarrow b$
IC $\emptyset$
Ab a, b
G p

The initial state is $\langle \emptyset, \emptyset, \epsilon, \emptyset, p\rangle$ and the transition sequence starts by applying a step with the (*Cla*)-rule for r_1. This step is possible because p belongs to $\mathfrak{D}$ and there does not exist a non-checked, non-checking IC, $i \equiv (L \rightarrow R)$, such that $p \in defs(L)$. The state obtained is $\langle \emptyset, \emptyset, \epsilon, \emptyset, \neg q\rangle$ and the transition sequence follows with the application of the (*Naf*)-rule for r_2. Note that there are two possible options for which the condition of this rule can be satisfied, leading to two possible transition sequences and answers for this program: $\Delta = \{a\}$ and $\Delta = \{b\}$. Figure 2 details the steps of one of these transition sequences.

By Definition 3.2, the abducible set $\{a\}$ in the tuple $\langle \{true \rightarrow r\}, \{a\}, \epsilon, \{true \rightarrow r\}, true\rangle$ directly provides the answer for this program and goal, since it is not necessary to check any additional constraint.

As commented before, the condition for applying the (*Naf*)-rule to r_2 requires evaluating $Sat(\{true \rightarrow r\}, \{true \rightarrow r\}, \emptyset, \epsilon, \emptyset)$, which can also output $\langle\{true \rightarrow r\}, \{b\}, \{true \rightarrow r\}\rangle$ if in step (*Abd.1*) rule r_4 is selected instead of r_3. In such a case, the final answer $\Delta = \{b\}$ is obtained. □

4 Compiling PIFFC programs

This section provides a compilation approach to solving ALP goals following the operational semantics just described. The rationale behind this compilation approach is to have a Prolog program that represents the possible derivations for a goal ϕ under a program, explicitly including triggerings of ICs (in particular, from the *Sat* function) and the program transformation result of *transform*. Such transformations and generic calls to *Sat* are thus avoided thanks to the precompilation. The result of executing ϕ are the assumed abducibles for each answer.

Given a goal ϕ and an ALP program $\Pi = \langle KB, IC, Ab\rangle$, the rules in KB and the ICs in IC are compiled into a Prolog program considering that the set of predicates Ab are the (undefined but assumable) predicates. The goal ϕ is also compiled as if it was a rule body. After consulting this Prolog program, all the answers to the (compiled) user goal ϕ can be interactively generated, analogously to typical Prolog systems. During solving, ICs are fired by either defined or abducible predicate calls (these latter being added to Δ). Note that any IC, i, may also fire any other IC that contains in its condition either predicates or abducibles present in the conclusion of i. If the conclusion of a given IC has been proven, there is no need to recheck the IC afterwards. Any IC that still needs to be verified is checked at the end of goal solving to ensure $KB \cup \Delta \models IC$. There are two reasons for an IC needing to be verified after goal solving: because either it has not already been fired, or its condition could not be proved when it was fired (eventually, the condition might be fully checked due to new assumptions made during further goal solving). Following the notation in Sect. 3, all these ICs are in $IC \setminus N$.

4.1 Textual syntax of ALP programs

Syntax of the formal ALP language as described in Sect. 2 is used in the next subsections to define the compilation. For the concrete syntax of ALP programs (which will be used in the implementation as the source language for the compilation), (Alberti et al. 2013) is adhered to, which to some extent follows Prolog syntax, where: an abducible a is written as `abd(a)`; defined predicates are not tagged (any predicate which is not in `abd`/1 is assumed to be defined); as negation `naf(`p`)` is considered, where p is either a defined or an abducible predicate; conjunction $\wedge$ (resp. disjunction $\vee$) corresponds to `','`/2 (resp. `';'`/2); and implication $\rightarrow$ is written as `'--->'`/2.

4.2 The *transform* function

As introduced in Sect. 3, this function is needed to handle the negation of a predicate. Instead of applying transition rules for rewriting $\neg p$ into a conjunctive formula in which no negation occurs in an IC condition, as in (Fung and Kowalski 1997), this procedure is left to the *transform* function in the operational semantics just presented in Sect. 3. This can even be enhanced in the system implementation if the original program is rewritten following the

transform steps. To this end, $\neg p$ is first transformed into $p \rightarrow false$; in a second step, p is successively unfolded such that all negations in its derivation sequence occur in place of p. To apply such an unfolding, a predicate dependency graph (PDG) is created after reading the user ALP program $\langle KB, IC \rangle$ to compile it.

For each rule $l \leftarrow r$ in KB, any positive literal conjunct c in r produces the arc $(l, c, +)$ as an element of the so-called dependency relation, meaning that l *positively* depends on c. The PDG is the transitive closure of the dependency relation. If c is a negative literal, $(l, c, -)$ is added instead, meaning that l *negatively* depends on c. A positive literal p must be unfolded if it depends on a negative literal. This is determined by inspecting whether $(p, c, -)$ belongs to the PDG. A literal that does not depend on negative literals is not unfolded. Since a predicate can be defined by more than one rule, the result of the unfoldings of a single IC may lead to several ICs.

Computing the transitive closure of a relation has time complexity $\mathcal{O}(n^3)$, which poses a significant overhead for compiling large programs, such as those introduced in Sect. 6. Since this closure is used to determine whether a negated call is reached by a given predicate call, it can be computed more efficiently by starting from each negated call. This involves traversing the direct arcs in the PDG from the conjunct to the head, and adding each head found during the traversal as a predicate that depends on a negated call. During this traversal, visited predicates are memorized to prevent non-termination in the event of a cycle.[3] Also, the traversal stops when a predicate that has been found to depend on a negated call is revisited.

Example 7 Unfolding $p \rightarrow false \in IC$ in the context of $KB = \{p \leftarrow q.p \leftarrow \neg r.q \leftarrow s.\}$ leads to two ICs: $q \rightarrow false$ and $\neg r \rightarrow false$ since the PDG became $\{(p, q, +), (p, r, -), (q, s, +), (p, s, +)\}$. Thus, note that q is not unfolded in the first resulting IC.

In the implementation, successive applications of unfolding steps are defined in the predicate `unfold`/2, and the unfold condition in `reaches_naf`/1. This dynamic predicate contains each labeled arc for each rule read from the ALP user program, plus all the arcs computed by the transitive closure, which is defined by the predicate `set_reaches_naf`/0. The third step in *transform* is a source-to-source transformation of the result of the second step by which the negative literals in the condition of an IC are rewritten as a disjunction in the head of the IC, as described in the Sect. 3. Following Example 7 above, $\neg r \rightarrow false$ is rewritten as $true \rightarrow false \vee r$, which is simplified to $true \rightarrow r$. The result of the transformation is implemented in the predicates `'$naf_p'`/0 and `'$naf_a'`/0 for defined and abducible predicates, respectively, which will be explained in Sects. 4.7 and 4.9.

4.3 Translating ICs to be checked after goal solving

Checking ICs after goal solving is the condition to be satisfied by following Definition 3.2. In this stage, each IC must be checked, if this has not been done already by some predicate call. Each IC $c_1 \wedge \ldots \wedge c_n \rightarrow D_1 \vee \ldots \vee D_m$ receives an identifier id. Relevant solutions include each D_i only if the condition holds. If the condition does not hold, then IC holds vacuously (with only one answer). Thus, a first attempt to implement the check of an IC is the predicate `'$ic'`/2:

[3] In any case, if this situation occurs, a goal depending on a cycle will also suffer from non-termination at runtime.

```
'$ic'(id, ps) :-
    (c'_1, ..., c'_n
     -> (D'_1; ...; D'_m)
     ;  true).
```

where *id* is the IC identifier (one id for each IC), *ps* is the list of predicates the IC condition depends on. Each c'_i and each d'_j (conjunct in D'_k) is translated as explained later. The list of firing predicates *ps* is intended as an output parameter, and it is needed by predicates invoking `'$ic'` to check whether any of its firing predicates has already been solved (if so, the IC must be checked).

The identifier *id* is an argument to ease finding all those ICs that have not been fired at the end of goal solving, which is implemented as follows:

```
findall(P, '$fic'(P), Ps),
nfsetof('$ic'(Id, FPs),
      Body^FP^
      (clause('$ic'(Id, FPs), Body),
       member(FP, FPs),
       \+ member(FP, Ps)),
      ICs),
maplist(call, ICs).
```

Here, each fact `'$fic'`/1 denotes in its argument a predicate that fires at least one IC and that was attempted to be solved. Each fact `'$fic'(P)` is asserted once upon solving `P` for the first time (cf. Subsect. 4.6). `nfsetof`/3 is a non-failing `setof`, simply defined as:

```
nfsetof(X, Y, Z) :-
    setof(X, Y, Z)
    -> true
    ;  Z = [].
```

Here, the if-then-else construct (*Condition* -> *TrueBranch*; *FalseBranch*) was used, which allows neater formulations in general (being more noticeable in forthcoming code excerpts). Note that the condition removes choice points, so that the above is equivalent to:

```
nfsetof(X, Y, Z) :-
    setof(X, Y, Z),
    !.
nfsetof(_X, _Y, []).
```

Though the if-then-else construct is mainly used throughout the article, the equivalent formulation is now employed in the system implementation, which can be downloaded from Gavanelli et al. (2024).

Recall that abducibles occurring in IC conditions must not be assumed, only checked. If c_i is an abducible, it can be compiled simply to a test over c_i (as later described in Sect. 4.5.1). If c_i is a call to a defined predicate, assumptions due to this predicate must be disallowed (note that this predicate might perform assumptions, either when solving its body or as a result of IC triggering). Thus, assumptions are disabled when checking an IC condition as follows:

```
'$ic'(id, ps) :-
    (('$no_assumptions'
        ->  c'_1, ..., c'_n
        ;   b_assertz('$no_assumptions'),
```

```
            c'_1, ..., c'_n,
            retract('$no_assumptions'))
     -> (D'_1; ...; D'_m)
     ;  true
    ).
```

Here, `'$no_assumptions'/0` is the fact used to block assumptions, i.e., the implementation device corresponding to adding the witness element $\mathcal{B}$ to Δ. It is asserted whenever an IC condition is to be checked, but only once because further ICs can be involved when solving this checking. The fact is retracted by the responsible IC when checking is done, either explicitly (with `retract/1`) or implicitly in the presence of failure (with `b_assertz/1`). This is the implementation counterpart for removing the witness element from Δ.

The call to `b_assertz/1` corresponds to a backtrackable assertion, as defined by (Wielemaker 2022), which avoids leaving a choice point open:

```
b_assertz(Term) :-
    assertz(Term, Ref),
    undo(erase(Ref)).
```

`Term` is asserted when `b_assertz` is called. Upon backtracking, the assertion is undone (i.e., `Term` is automatically retracted). Note that the less common `assertz/2` is employed here to retrieve the reference to the asserted term (`Term`), which is used to erase it upon backtracking by `undo/1`. As explained in Wielemaker (2022), this is equivalent to the more portable code:

```
b_assertz(Term) :-
    assertz(Term, Ref),
    (   true
    ;   erase(Ref),
        fail
    ).
```

But `undo/1` has the advantage of avoiding leaving an open choice-point, and is therefore more efficient.

Additionally, the translation of ICs can be enhanced by caching successful ICs as follows:

```
'$ic'(id, ps) :-
    ('$ic_checked_id'
        -> true
        ;   (('$no_assumptions'
                -> c'_1, ..., c'_n
                ;  b_assertz('$no_assumptions'),
                   c'_1, ..., c'_n,
                   retract('$no_assumptions'))
             -> (D'_1; ...; D'_m),
                b_assertz('$ic_checked_id')
             ;  true)).
```

The fact `'$ic_checked_id'/0` indicates that the IC with identifier `id` has already been checked and does not need to be reevaluated, which corresponds to elements in the set I in conditions of the transitions rules, such as $\forall i_k \equiv (L_k \rightarrow R_k) \in IC \backslash I$, in the operational semantics (see Sect. 3).

Note that the default case of `'$ic'/2` is simply `true` and it does not include the backtrackable assertion of '$ic_checked_*id*' because a failed condition does not necessarily mean that the IC vacuously holds. Indeed, it may be the case that a given abducible present in the condition has not been assumed yet when checking this condition, and another further IC may assume it.

Also note that this caching is not appropriate for ICs with a false conclusion (for example, those coming from negated goals) because, clearly, the IC cannot be checked as true. In this case, the translation is simplified to:

```
'$ic'(id, ps) :-
    (('$no_assumptions'
        -> c'_1, ..., c'_n
        ;  b_assertz('$no_assumptions'),
           c'_1, ..., c'_n
     -> false
     ;  true).
```

In this simplification, first, the code can be specialized by removing the call to `'$ic_checked_id'/0` (because the IC cannot be satisfied). And, second, there is no need even for `retract('$no_assumptions')` because if the condition succeeds, `'$no_assumptions'` will be automatically removed by `retract('$no_assumptions')` when reaching the `false` goal in the last but one line of `'$ic'/2`.

When the number of disjuncts D'_i is large, SWI-Prolog takes a significant time to parse such a large body. Thus, in both the above cases, the disjunction $(D'_1; \ldots; D'_m)$ is translated into a goal `'$ic_$r_l_id'` to the predicate defined by m clauses `'$ic_$r_l_id':-` D'_i.

Example 8 As an example of translating ICs to be checked after goal solving, see Example 5, which includes two integrity constraints:

IC i_1: *rain_last_night* $\rightarrow$ *cloudy_last_night*
i_2: *cloudy_last_night* $\rightarrow$ *false*
Ab *rain_last_night*, *sprinkler_was_on*, *cloudy_last_night*

Translation of these ICs becomes:

```
'$ic'(1,[rain_last_night]) :-
  ('$ic_checked_1'
   -> true
   ;  (('$no_assumptions'
         -> rain_last_night
         ;  b_assertz('$no_assumptions'),
            rain_last_night,
            retract('$no_assumptions'))
     -> '$abd_cloudy_last_night',
        b_assertz('$ic_checked_1')
     ;  true)).

'$ic'(2,[cloudy_last_night]) :-
  (('$no_assumptions'
```

```
    -> cloudy_last_night
    ;  b_assertz('$no_assumptions'),
       cloudy_last_night)
  -> false
  ;  true).
```

The first integrity constraint (with given identifier `id` = 1 as the second argument of the predicate) follows the generic translation, while the second one (with `id` = 2) follows the optimized translation due to its false conclusion.

4.4 Translating ICs to be checked during goal solving

Checking ICs during goal solving is requested by several transition rules (e.g., *Abd* and *Cla* in Sect. 3) in the call to the *Sat* function. Pruning computations in advance was the intention of checking ICs during goal solving, but this can be enhanced as follows. Each IC can be triggered every time a predicate call succeeds such that the predicate occurs in the IC condition. Thus, one compiled IC is generated for each call in the condition, removing the call in the new condition. If during goal solving an IC holds, it is marked as checked and it is not reevaluated afterwards (i.e., added to the set I in the operational semantics). Once a solution is found for the main user goal, further solutions due to alternatives in the IC can be obtained by backtracking. An IC:

$$c_1 \wedge \ldots \wedge c_{i-1} \wedge l \wedge c_{i+1} \wedge \ldots \wedge c_n \rightarrow D_1 \vee \ldots \vee D_m$$

is translated into n predicates `'$ic_`*l*`_`*id*`'`/0, one for each literal l in the body of the IC:

```
'$ic_l_id' :-
    ('$ic_checked_id'
        -> true
        ;   (('$no_assumptions'
                -> c'_1, ..., c'_{i-1}, c'_{i+1}, ..., c'_n
                ;  b_assertz('$no_assumptions'),
                   c'_1, ..., c'_{i-1}, c'_{i+1}, ..., c'_n,
                   retract('$no_assumptions'))
            ->  (D'_1; ...; D'_m),
                b_assertz('$ic_checked_id')
            ;   true)).
```

where l includes either a defined or an abducible predicate.

A particular case of this predicate is when either the condition is empty (i.e., containing only a literal which is removed in the compilation) or the condition is `true`. A call to this IC assumes that the condition has already been satisfied, so that there is no default case. In addition, handling of `'$no_assumptions'` is therefore unneeded and thus removed:

```
'$ic_l_id' :-
    ('$ic_checked_id'
        ->  true
        ;   (D'_1; ...; D'_m),
            b_assertz('$ic_checked_id')).
```

In a similar way to translating ICs to be checked after goal solving, another particular case is when the conclusion of an IC is false, so the code can be specialized first by removing the call to `'$ic_checked_id'/0` and, second, by removing `retract('$no_assumptions')` as follows:

```
'$ic_l_id' :-
    (('$no_assumptions'
         -> c'_1, ..., c'_{i-1}, c'_{i+1}, ..., c'_n
         ;  b_assertz('$no_assumptions'),
            c'_1, ..., c'_{i-1}, c'_{i+1}, ..., c'_n)
     -> false
     ;  true).
```

If, in addition, the condition is empty, it is simplified to:

```
'$ic_l_id' :-
    false.
```

because handling of `'$no_assumptions'` is not needed as indicated before. Note that a clause with a false body is not equivalent to removing the clause, because in this last case, when solving the goal `'$ic_l_id'`, SWI-Prolog would raise an exception, instead of just failing, which is what is sought.

Example 9 Recalling again the two ICs in Example 5, the following translation is obtained:

```
'$ic_rain_last_night_1' :-
  ('$ic_checked_1'
      -> true
      ;  '$abd_cloudy_last_night',
         b_assertz('$ic_checked_1')).

'$ic_cloudy_last_night_2' :-
  false.
```

The first translated IC (with `id` = 1 in the name of the predicate) follows the particular case when the condition is empty because it only has the literal `rain_last_night/0` in its condition, which is removed because it does not need to be checked, since the predicate `rain_last_night/0` triggers the IC (so it is already checked). The second translated IC (with `id` = 2) simply becomes a failing predicate because it has an empty antecedent (for the same reason as IC 1) and its conclusion is false.

4.5 Translating conditions and conclusions of ICs

Next, translations of conjuncts c_i and disjuncts D_i in ICs are described.

4.5.1 Conjuncts in implication conditions

Each c_i is translated as follows: First, calls to abducibles are no longer annotated in the textual syntax as they are in the user ALP program. Second, negated predicates (either defined or abducibles) are removed from IC conditions by rewriting the original formula (just as in the third step of *transform*):

$$c_1 \wedge \ldots \wedge c_i \wedge \neg c_{i+1} \wedge \ldots \wedge \neg c_n \rightarrow D_1 \vee \ldots \vee D_m$$

Table 1 Translating conjuncts in IC conclusions and clause bodies

c_i	c_i'	Condition
p	p	$\nexists$'$ic_p_id'
	'$p'	$\exists$'$ic_p_id'
$\neg p$	'$naf_p'	
a	'$abd_a'	
$\neg a$	'$naf_abd_a'	

into the semantically equivalent one:

$$c_1 \wedge \ldots \wedge c_i \rightarrow D_1 \vee \ldots \vee D_m \vee c_{i+1} \vee \ldots \vee c_n$$

so that negation can only occur at IC conclusions and rule bodies. But if the result of this rewriting leads to a negated predicate, a specific predicate call to handle the negation is generated, in a similar way to how the function *transform* operates: replacing negation by IC satisfaction (cf. next Sect. 4.5.2).

4.5.2 Conjuncts in conclusions

Each conjunct c_i in each disjunct D_j for an IC with identifier `id` is translated into c_i' as shown in Table 1, where `'$ic_p_id'`/0 is a fact which means that there is an IC with identifier `id` which is triggered by p, where p can be either a defined or an abducible predicate. For the rows in the table with negated literals, there are no conditions on triggered ICs because the corresponding c_i' translations correspond to predicates (`'$naf_p'` and `'$naf_abd_a'`) managing the transformation of negated predicates, which are responsible for firing the needed ICs. The condition in the third column of this table is tested by the compiler with facts `'$triggered_by'(id, call)`, where `id` is the IC identifier and `call` is either a defined or an abducible predicate. This fact is asserted in a first compilation stage for each predicate occurring in the condition of each IC.

In the first case of the table, a predicate call p is simply translated to the call `p` when p does not occur in the condition of any IC, or to `'$p'` otherwise, which will trigger the necessary ICs. The second case corresponds to $\neg p$, which is translated into `'$naf_p'`, dealing with processing the negation and is described later in Subsect. 4.7. The third case is a call to an abducible a, which can always be assumed and succeed, unless an IC with `a` in its (succeeding) condition leads to a failing conclusion (this is processed with the predicate `'$abd_a'`/0 described later in Sect. 4.8). In the last case, the negation of an abducible $\neg a$ is translated into the call `'$naf_abd_a'`, which deals with processing the negation of the abducible and is described later in Subsect. 4.9.

4.6 Compiling defined predicates with associated ICs

If a defined predicate p occurs in the condition of some IC, the atom p is replaced by `'$p'`, which calls the n ICs triggered by p; the predicate `'$p'`/0 is defined as:

```
'$p' :-
    assertz_once('$fic'(p)),
    p,
    '$ic_p_1',
    ...
```

`'$ic_p_n'.`

`assertz_once`/1 asserts its argument only if it was not asserted before (multiple calls to `'$p'` result in a single asserted fact `'$fic'(p)`). Before a predicate `p` is called, the fact `'$fic'`/1 is asserted. Recall that a fact `'$fic'(p)` means that an attempt was made to solve `p` and its corresponding ICs and, after goal solving, ICs depending on `p` should not be retriggered after the proof solely due to `p`. So, `'$fic'`/1 facts are added with non-backtrackable assert, as they should not be automatically removed in order to avoid as many retriggerings as possible. These `'$fic'` facts are removed at the start of goal solving.

If there is a fact `'$fic'(p)`, ICs sensitive to p will not be fired solely due to p at the end of goal solving (nonetheless, other predicates may fire them). If `'$p'` succeeds, the corresponding ICs for p have been checked with success.

Possible cases with respect to IC checking at the end of goal solving are:

- If `p` fails, no IC is checked. No IC condition will succeed: no need to check conclusions at the end.
- If `p` succeeds, its ICs are sequentially checked. Possible cases:
  - All ICs succeed (all ICs have been checked and do not need to be rechecked at the end).
  - `'$ic_p_i'` fails and no subsequent IC is tried. If one IC fails, the current solution is discarded and thus no final IC checking will be needed for this branch.

Thus, in any case, these ICs do not need to be rechecked for a possible failure at the end of goal solving.

In addition, the call to `'$p'` can be optimized by caching it by means of the flag `'$solved'`/1 with a predicate name as its argument, as follows:

```
'$p' :-
    ('$solved'(p)
         -> true
         ;  assertz_once('$fic'(p)),
            p,
            b_assertz_once('$solved'(p)),
            '$ic_p_1',
            …
            '$ic_p_n').
```

This also avoids non-termination due to interdependent ICs, as shown in the next example:

Example 10 Consider the following program rules, ICs and goal:

$$
\begin{array}{ll}
KB & r_1 : p \leftarrow true \\
 & r_2 : q \leftarrow true \\
IC & i_1 : p \rightarrow q \\
 & i_2 : q \rightarrow p \\
G & p
\end{array}
$$

Translating this program gives the following results for the defined predicates `p` and `q`:

```
'$p' :-
  ('$solved'(p)
    -> true
    ;  assertz_once('$fic'(p)),
       p,
       b_assertz_once('$solved'(p)),
       '$ic_p_1').

'$q' :-
  ('$solved'(q),
    -> true
    ;  assertz_once('$fic'(q)),
       q,
       b_assertz_once('$solved'(q)),
       '$ic_q_2').
```

where the ICs are compiled as explained in Sect. 4.4 as follows:

```
'$ic_p_1' :-
  ('$ic_checked_1'
    -> true
    ; '$q',
      b_assertz('$ic_checked_1')).

'$ic_q_2' :-
  ('$ic_checked_2'
    -> true
    ;  '$p',
       b_assertz('$ic_checked_2')).
```

4.7 Compiling the negation of defined predicates

As explained in the operational semantics, a negative literal $\neg p$ is translated into the semantically equivalent implication $IC \equiv p \rightarrow false$; in the implementation, each call $\neg p$ is compiled to a call to the predicate `'$naf_p'`/0. Such a predicate takes care of triggering the IC, which itself is compiled as described in Subsect. 4.3 for the ICs written by the user, with a caveat: since all ICs (that have not been triggered) are tested at the end of the derivation, and since the ICs are compiled, there would be a risk to wrongly execute also the ICs deriving from a rewritten negation; clearly such ICs must be triggered only if the respective negative goal has been invoked. For this reason, ICs derived from a negative goal must be enabled and predicate `'$naf_p'`/0 first enables the IC, then triggers it:

```
'$naf_p' :-
    b_assertz_once('$ic_enabled_id'),
    '$ic(id, _)'.
```

where `b_assertz_once`/1 is analogous to `assertz_once`/1 but for the backtrackable assertion, and *id* is the identifier for IC, which is compiled as:

```
'$ic(id, [p])' :-
```

```
    ('$ic_enabled_id'
        ->  IC compilation as of Subsection 4.3
        ;   true).
```

Example 11 Another variation of Example 5 may be considered as an example of translating the negation of a defined predicate:

KB r_1: *grass_is_dry* $\leftarrow \neg$*wet_condition*
r_2: *wet_condition* $\leftarrow$ *rain_last_night*
r_3: *wet_condition* $\leftarrow$ *sprinkler_was_on*
Ab *rain_last_night*, *sprinkler_was_on*

Translation of r_1 becomes:

```
grass_is_dry :- '$naf_wet_condition'.

'$naf_wet_condition' :-
  b_assertz_once('$ic_enabled_1'),
  '$ic'(1,_).

'$ic'(1,[wet_condition]) :-
  ('$ic_enabled_1'
    -> (('$no_assumptions'
         -> wet_condition
         ;  b_assertz('$no_assumptions'),
            wet_condition)
        -> false
        ;  true)
    ; true).
```

4.8 Compiling abducible predicates

Each call to an abducible predicate a checks whether the abducible has already been assumed. If so, its corresponding ICs have succeeded and nothing is left to do. Otherwise, if assumptions are not blocked (cf. Subsect. 4.3), it is asserted and calls to the n ICs that are triggered by a are added:

```
'$abd_a' :-
    (a   -> true
         ;  \+ '$no_assumptions',
            b_assertz(a),
            '$ic_a_1',
            …
            '$ic_a_n').
```

Here, the negated call `\+ '$no_assumptions'` allows for disabling assumptions should the current solving path correspond to checking an IC condition.

Example 12 Following Example 5 again, there are three abducibles (*rain_last_night*, *sprinkler_was_on*, and *cloudy_last_night*), each one receiving a compilation following this translation scheme (where the code for the last abducible is omitted because it is similar to the first):

```
'$abd_rain_last_night' :-
  (rain_last_night -> true
                   ;  \+ '$no_assumptions',
                      b_assertz(rain_last_night),
                      '$ic_rain_last_night_1').

'$abd_sprinkler_was_on' :-
  (sprinkler_was_on -> true
                    ; \+ '$no_assumptions',
                      b_assertz(sprinkler_was_on)).
```

While the first abducible can trigger an IC (`'$ic_rain_last_night_1'`), the second occurs in no IC.

4.9 Compiling the negation of abducible predicates

Similar to the compilation of the negation of defined predicates, a negative literal $\neg a$ is translated into the semantically equivalent implication $IC \equiv a \rightarrow false$. However, the implementation is simpler in this case, because if a has already been assumed, $\neg a$ should simply fail. And if otherwise $\neg a$ succeeds, further assumptions of a must be blocked. Thus, the call $\neg a$ is compiled to the predicate `'$naf_abd_a'`/0, which is implemented as follows:

```
'$naf_abd_a' :-
    (a   -> fail
         ;  b_assertz_once('$abd_a_false')).
```

Here, the fact `'$abd_a_false'`/0 represents blocking assumptions on a, i.e., a is assumed to be $false$. Thus, the corresponding IC does not need to be explicitly stated, but otherwise simply integrated into the compilation of the predicate `'$abd_a'`/0 of the abducible as follows:

```
'$abd_a' :-
    (a   -> true
         ;  \+ '$abd_a_false',
            \+ '$no_assumptions',
            b_assertz(a)),
    '$ic_a_1',
    …
    '$ic_a_n'.
```

Example 13 Consider a modified version of Example 5, where it is not known whether it was cloudy last night:

$KB \quad r_1$: *grass_is_wet* $\leftarrow$ *rain_last_night*

r_2: *grass_is_wet* ← *sprinkler_was_on*
IC i_1: *rain_last_night* → ¬*cloudless_last_night*
Ab *rain_last_night*, *sprinkler_was_on*, *cloudless_last_night*
G *grass_is_wet*

The negated abducible ¬*cloudless_last_night* is therefore translated as:

```
'$naf_abd_cloudless_last_night' :-
  (cloudless_last_night -> fail
;  b_assertz_once('$abd_cloudless_last_night_false')).
```

and the abducible itself is translated as:

```
'$abd_cloudless_last_night' :-
  (cloudless_last_night -> true
;\+ '$abd_cloudless_last_night_false',
 \+ '$no_assumptions',
 b_assertz(cloudless_last_night)).
```

In this case, there is no IC triggered by *cloudless_last_night*.

Compilations of abducibles which are not present in negated calls are compiled as in Subsect. 4.8.

4.10 Translating rules

A rule $h \leftarrow c_1 \wedge \ldots \wedge c_n$, where each c_i is a literal, is translated into: h :- c'_1, ..., c'_n, where each c'_i is the translation of c_i following Table 1, in the same way that conjuncts in conclusions of ICs are translated. Additionally, for the convenience of the user, disjunctions in body rules $h \leftarrow d_1 \vee \ldots \vee d_n$ are allowed in the implementation, and are automatically converted as a source-to-source transformation into n alternative rules: $h \leftarrow d_1, \ldots, h \leftarrow d_n$.

Example 14 Program rules in Example 1 are simply translated into:

```
grass_is_wet :- '$abd_rain_last_night'.
grass_is_wet :- '$abd_sprinkler_was_on'.
```

5 Testing the correctness of the implementation

This section leverages the differential testing technique to compare the operational semantics and its implementation with respect to an oracle, this being the SCIFF system, a correct implementation of ALP. While comparing the outcomes of two implementations (say SCIFF and PIFFC) for the same inputs is a straightforward task, comparing an operational semantics is not. Thus, in this work an implementation has been developed of a proof system (called PTS: Propositional Transition System) which is able to reproduce derivations such as the one depicted in Example 5. Computed answers from these proofs can then be effectively compared to the result of the system implementations. This implementation and test programs can be downloaded from Gavanelli et al. (2024).

5.1 Differential testing

Differential testing (McKeeman 1998) refers to a technique that compares the outputs or behaviors of different software implementations to detect discrepancies or potential bugs. It involves running the same test suite on multiple implementations and analyzing the differences in their outputs or behaviors.

Typically, it works in three phases: i) Test Generation, where a test suite is generated either automatically or manually; ii) Test Execution, in this phase the generated test suite is used to feed two or more different implementations of the system being tested; iii) Comparison, in this last phase, the outputs or behaviors of the different implementations obtained in the former phase are compared; a difference in the outputs indicates a discrepancy or potential bug between the implementations; iv) Analysis and Bug Detection, when discrepancies are found, they must be analyzed to determine whether they are real bugs that should be reported and addressed by the software developers.

This technique has been successfully applied in various domains, including compilers (McKeeman 1998; Yang et al. 2011; Le et al. 2014), virtual machines (Chen et al. 2016), network protocols (Chen and Su 2015), and more (Petsios et al. 2017). It can uncover subtle (semantic) bugs, security vulnerabilities, and compatibility issues that may go unnoticed with traditional testing methods.

It is fair to note that, as with other testing techniques, differential testing can only increase the degree of confidence in the quality of software but cannot determine its correctness. In the present case, it can increase the confidence in the semantic equivalence of two systems, if all the tests performed are positive, or, on the contrary, to detect semantic discrepancies, in the event that a test fails.

5.2 Comparing the implementations of the systems PTS, PIFFC and SCIFF

It is also common to compare implementations under development with another, more mature, implementation of a target system (or specification) which is considered to be an oracle. Therefore, in general, it is a technique used to increase the quality of complex software systems.

In the present case, we have three systems which are intended to implement the same declarative semantics: PTS, PIFFC and SCIFF. PTS is the direct implementation of the operational semantics as described in Sect. 3, which is intended to output the proof(s) for an input goal under a program and a set of ICs; i.e., it is intended to automate the process of generating proofs. Its display is similar to what is shown in Fig. 1: Each transition step in the output is labeled with the applied rule, and the result for transition rules and conditions is displayed as the output state (an output state followed by the same input state for the next rule is shown as only one state). A final state with a true goal is an answer, and every answer for a given input goal is obtained by backtracking. Two components of the state are key when checking the declarative semantic equivalence: assumed abducibles and final goal.

The SCIFF system is taken as the canonical system and, therefore, as the oracle to which the others have to conform. The idea is to follow the aforementioned steps: i) to create a complete set of abductive programs exercising every corner of the SCIFF system; ii) to execute the set of abductive programs in both PTS and PIFFC systems and in the SCIFF system and to see if the results match, assuming that the answer of the SCIFF system is the correct one.

The differential testing is a correctness check that systems PTS and PIFFC are in accordance with the SCIFF system; if the test suite is complete enough and no differences are appreciated, the confidence that the three systems are in agreement is corroborated. Recall that the SCIFF system implements transition rules with CHR rules (Alberti et al. 2022), reducing the gap between the IFF operational semantics (Fung and Kowalski 1997) and its implementation. Moreover, it could also be said that the operational semantics defined in this article is an operational semantics oriented towards an efficient implementation (cf. Sect. 6), and that it conforms to the semantics of the SCIFF system.

As a test suite, a total of more than 90 tests were selected, which are executed in SCIFF, PIFFC and PTS, and use every single transition rule in the operational semantics. A test coverage is implemented in a script that checks whether all transition rules are used in the test programs. All these tests were passed and no rule was unused, therefore achieving the intended goals of differential testing.

6 Performance

This section analyses the feasibility of the proposal by examining its performance on different parametric benchmarks developed to mainly assess its scalability with respect to using abducibles and integrity constraints. In order to have a fair performance comparison, SWI-Prolog-based systems which do support sound negation were selected.[4] The following two systems which are available online and run in SWI-Prolog (Wielemaker et al. 2012) were considered: SCIFF (Alberti et al. 2022), and SCASP (Arias et al. 2018), in addition to the compiled proposal in this study (referred to as PIFFC). With respect to the test programs, the focus is on testing intensive use of abducibles and integrity constraints. Thus, the following parametrized tests are proposed (on a size n) on the number of abducibles, alternatives, and integrity constraints so their size can be scaled up and their scalability performance tested.

b01 Creates a chain of n assumptions with abducibles $\mathtt{a}_i$ by invoking n predicates: $KB = (\bigcup_{1\leq i<n} \{(\mathtt{p}_i \mathtt{:-\ abd(a}_i\mathtt{),\ p}_{i+1}\mathtt{)}\}) \cup \{(\mathtt{p}_n \mathtt{:-\ abd(a}_n\mathtt{)})\}$, $IC = \emptyset$, $\phi = \mathtt{p1}$. This program takes the parameter n to test a single derivation including an explanation of n abducibles.

b02 Creates n abducible alternatives for a predicate $\mathtt{p}$. $KB = \bigcup_{1\leq i\leq n} \{(\mathtt{p:-\ abd(a}_i\mathtt{)})\}$, $IC = \emptyset$, $\phi = \mathtt{p}$. *b02* is thus intended to check n alternatives of a single explanation.

b03 Attach an IC with a conjunction of n abducibles in its conclusion to a predicate $\mathtt{p}$ defined by a single fact: $KB = \{(\mathtt{p})\}$, $IC = \{\mathtt{p\ ---\!>\ abd(a}_1\mathtt{)} \wedge \cdots \wedge \mathtt{\ abd(a}_n\mathtt{)}\}$, $\phi = \mathtt{p}$. *b03* checks the performance of an IC with n conjuncts.

b04 Attach an IC with a disjunction of n abducibles in its conclusion to a predicate $\mathtt{p}$ defined by a single fact: $KB = \{(\mathtt{p})\}$, $IC = \{\mathtt{p\ ---\!>\ abd(a}_1\mathtt{)} \vee \cdots \vee \mathtt{abd(a}_n\mathtt{)}\}$, $\phi = \mathtt{p}$. Similar to *b03*, *b04* checks the performance of an IC with n disjuncts

b05 Similar to *b01*, but creating a single abducible: $KB = \bigcup_{1\leq i<n} \{(\mathtt{p}_i \mathtt{:-\ abd(a),\ p}_{i+1}\mathtt{)}\} \cup \{(\mathtt{p}_n \mathtt{:-\ abd(a)})\}$, $IC = \emptyset, \phi = \mathtt{p}_1$. In *b05*, the same explanation (instead of n abducibles) is given for the n rules, therefore checking repeated testing on an existing abducible.

b06 Creates a chain of n abducibles by triggering $n-1$ integrity constraints: $KB = \bigcup_{1\leq i\leq n} \{(\mathtt{p}_i \mathtt{:-\ abd(a}_i\mathtt{)})\}$, $IC = \bigcup_{1\leq i<n} \{\mathtt{abd(a}_i\mathtt{)\ ---\!>\ p}_{i+1}\}$, $\phi = \mathtt{p}_1$. Similar to *b01*, this program takes the parameter n to test a single derivation including an explana-

[4] HYPROLOG (Christiansen and Dahl 2005) does not support sound negation.

tion of n abducibles, but this time by means of triggering integrity constraints instead of a chain of predicate calls.

b07 Tests chained predicate calls with no ICs: $KB = \bigcup_{1 \leq i < n} \{(\mathtt{p}_i \mathtt{:-}\ \mathtt{p}_{i+1})\} \cup \{(\mathtt{p}_n)\}$, $IC = \emptyset$, $\phi = \mathtt{p}_1$. This program tests a propositional program with n calls and with neither integrity constraints nor abducibles.

b08 Tests chained negated predicate calls with no ICs: $KB = \bigcup_{1 \leq i < n} \{(\mathtt{p}_i \mathtt{:-}\ \mathtt{naf(p}_{i+1}\mathtt{))}\} \cup \{(\mathtt{p}_n)\}$, $IC = \emptyset$, $\phi = \mathtt{p}_1$. This program is similar to *b07*, but including n negated calls.

b09 Tests chained negated abducible predicate calls with ICs: $KB = \bigcup_{1 \leq i < n} \{(\mathtt{p}_i \mathtt{:-}\ \mathtt{naf(abd(a}_i\mathtt{))},\ \mathtt{p}_{i+1}), (\mathtt{p}_n \mathtt{:-}\ \mathtt{naf(abd(a}_n\mathtt{)))}\}, IC = \bigcup_{1 \leq i \leq n} \{\mathtt{abd(a}_i\mathtt{)}\ \mathtt{--->}\ \mathtt{false}\}$, $\phi = \mathtt{p}_1$. This program is similar to *b09*, but including an IC for each one of the n abducibles, making the negation succeed.

There are three stages in both systems SCIFF and PIFFC: compilation of the source ALP program to Prolog files, consulting these compiled files into the SWI-Prolog system, and execution of the user ALP goal. Thus, the amounts of time required for each of these three stages are considered as time measurements, needed for solving the goal ϕ in each test program. In addition, the size of the generated Prolog files are included in the comparison to study the trade-off between performance and space requirements. Unfortunately, SCASP was not fast enough to be included in the comparison (for example, in test program *b01*, 11 s were needed to solve a small instance with $n = 200$).

An Intel Xeon CPU E3-1505 M v5 with 4 physical cores at 2.8 GHz, 8 threads, 16GiB RAM, with the Windows 10 64-bit operating system is used as a test platform. Test programs are run on the development version of SWI-Prolog 64-bit 9.1.17-13-g7f0b8ab07 (as of 24/10/2023, which fixes a bug in the undo operation used by `b_assertz`). Execution times of goals correspond to all solutions, are given in seconds and are the result of averaging 10 runs (after discarding the first because the OS seems to waste time allocating resources for calling SWI-Prolog and spend more time running the test; smaller standard deviations were observed when considering only runs from the second onward). These times were measured with the built-in predicate `statistics/2` and its key cputime for SWI-Prolog, which returns the total run-time measurement.

Table 2 includes the following columns (times in seconds): Test (test name), n (size of the test), System (name of the system), Size (file size in KB), CTime (compilation time), LTime (loading time), ETime (execution time of all answers), ×Size (ratio of PIFFC and SCIFF file sizes), ETime^{SU} (execution time speed-up, i.e., ratio of SCIFF and PIFFC execution times), and TTime^{SU} (total time speed-up, i.e., ratio of SCIFF and PIFFC times of the three stages – compilation, loading and execution). These three last columns are calculated by using the actual data with all decimal places as computed by SWI-Prolog, while data displayed in the table are limited to three decimal places.

The column ETime^{SU} shows notable speed-ups of up to 2002× for PIFFC over SCIFF; in particular, for those which create long chains of assumptions (*b01*-2002×, *b03*-524×, and *b06*-449×). Speed-ups for goals involving disjunctions (*b02* and *b04*) are not so notable, though nevertheless important (2.594× and 1.664×). Test programs involving negation (*b08* and *b09*) also display speed-ups from 2.160× to 6.055. Though the test *b07* displays a minor speed-up, note that execution times in both systems are so tiny (a few milliseconds) that they make no difference in the end: Consider that the compilation time in SCIFF for this test is 402.563 seconds compared to only 1.078 in PIFFC (373.435× faster). In addition, observe that file size and loading time are better in PIFFC.

With respect to the sizes of the compiled files for both systems, in general, more space consumption is expected for PIFFC, ranging from 3.072× up to 18.121×. However, in

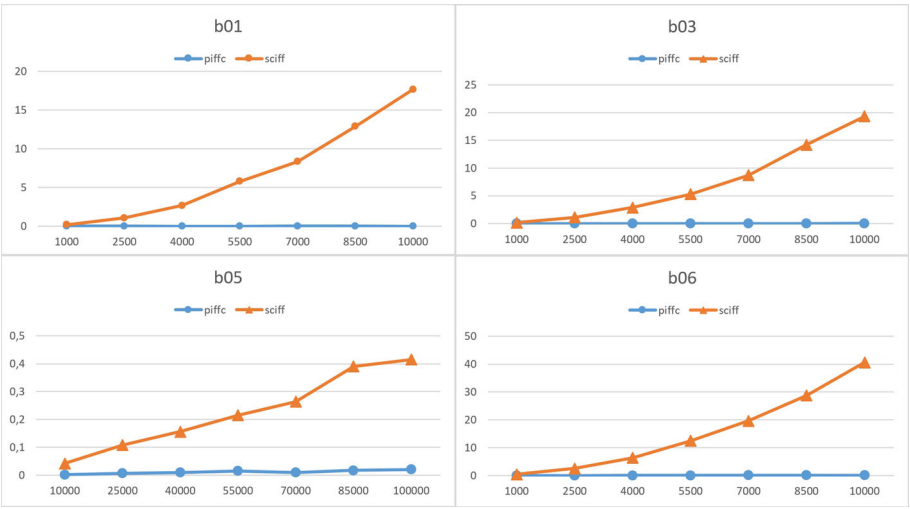

Fig. 3 Execution time as a function of the test size n

Table 2 Performance data

Test	n	System	Size	CTime	LTime	ETime	×Size	ETime^{SU}	TTime^{SU}
b01	10000	*piffc*	1967	0.438	1.219	0.009	3.072	2002.333	14.084
		sciff	640	4.172	0.516	18.772			
b02	100000	*piffc*	19450	2.906	11.922	0.708	7.407	2.594	0.337
		sciff	2626	1.328	2.078	1.836			
b03	10000	*piffc*	1930	0.297	1.078	0.036	18.121	524.783	13.710
		sciff	106	0.391	0.094	18.859			
b04	100000	*piffc*	20329	2.547	11.813	1.303	8.714	1.664	0.485
		sciff	2333	4.578	0.844	2.169			
b05	100000	*piffc*	2718	1.578	1.797	0.020	0.430	22.154	121.748
		sciff	6315	407.406	5.516	0.450			
b06	10000	*piffc*	6544	1.172	2.891	0.098	6.876	449.778	12.176
		sciff	952	5.750	0.641	44.275			
b07	100000	*piffc*	1839	1.078	1.438	0.011	0.365	1.143	160.988
		sciff	5046	402.563	4.172	0.013			
b08	100000	*piffc*	38319	8.281	18.344	0.275	6.924	6.312	15.876
		sciff	5534	420.453	4.875	1.736			
b09	10000	*piffc*	6419	5.094	3.344	0.117	6.117	2.160	0.764
		sciff	1049	5.484	0.797	0.253			

some particular cases such as *b05* and *b07*, the requirements are lower: from 0.430× to 0.365×, which equates to 2.33× and 2.75× respectively less space consumption for PIFFC. As expected, loading times are proportional to program sizes.

However, from an overall point of view (i.e., considering the total time for compilation, loading and executing all solutions as depicted in column ×TTime), in general there are better times for PIFFC, with speed-ups ranging from 12.176× up to 160.988×. Only in the cases *b02* and *b04* (involving disjunctions) and *b09* (involving chained negations) does PIFFC behave worse, mainly due to loading the large compiled files.[5] Even so, execution times are better.

In order to assess speed-up scalability, Fig. 3 includes some selected benchmarks for which high speed-ups are achieved. Test program sizes are chosen by dividing the size n in Table 2

[5] However, be aware that after loading a program, it may be prompted with many queries.

into a horizontal scale of 7 points. The blue and orange lines, marked respectively with small solid circles and triangles, correspond to execution times in the column ETime in Table 2 for PIFFC and SCIFF (both with a vertical scale in the left axis). For all test programs, execution times reveal that speed-up grows with the program size n, so that even better speed-ups are expected for larger values of n.

7 Conclusions and future work

This article describes PIFFC, a compiled approach to solving propositional ALP programs. An operational semantics, a differential testing correctness, an implementation and an assessment of its applicability are given, showing the increased performance with respect to the target system SCIFF. As future work, the aim is to study the formal correctness between PIFFC semantics and SCIFF semantics, and also, to develop the natural extension of this work to the first-order predicate case. Recursion will play a significant role in this setting, needing a detailed analysis, mainly of its performance. In addition, the extension will require the use of constraint solvers in different domains. This will undoubtedly require access to constraint stores to keep track of relevant constraints when assuming and, therefore, asserting abducibles. Following earlier work on domain cooperation (e.g., (Estévez-Martín et al. 2009)), the concepts of bridges and inter-solver propagation may be incorporated in the hope of enabling a more powerful implementation of constraint domains than in existing approaches to ALP.

Funding Open access funding provided by Università degli Studi di Ferrara within the CRUI-CARE Agreement. This work was supported by the State Research Agency (AEI) of the Spanish Ministry of Science and Innovation under grant PID2019-104735RB-C42 (SAFER), and by the Comunidad de Madrid, under the grant S2018/TCS-4339 (BLOQUES-CM), co-funded by EIE Funds of the European Union, and 2022 Project "InSANE: Investigating Sparse Algorithms in the post von Neumann Era" CUP_E55F22000270001, funded by GNCS. This research was also funded by the Italian Ministry of University and Research through PNRR - M4C2 - Investimento 1.3 (Decreto Direttoriale MUR n. 341 del 15/03/2022), Partenariato Esteso PE00000013 -"FAIR - Future Artificial Intelligence Research" - Spoke 8 "Pervasive AI", funded by the European Union under the "NextGeneration EU programme".

Declarations

Financial and proprietary interests The authors have no financial or proprietary interests in any material discussed in this article.

References

Abdennadher S, Christiansen H (2001) An experimental CLP platform for integrity constraints and abduction. In: Larsen HL, Andreasen T, Christiansen H, Kacprzyk J, Zadrożny S (eds) Flexible Query Answering Systems. Physica-Verlag HD, Heidelberg, pp 141–152

Alberti M, Gavanelli M, Lamma E (2013) The CHR-based Implementation of the SCIFF Abductive System. Fundam. Informaticae 124(4):365–381. https://doi.org/10.3233/FI-2013-839
Alberti M, Gavanelli M, Chesani F (2022) The SCIFF Abductive Proof Procedure. System Implementation, available at http://lia.deis.unibo.it/sciff/
Arias J, Carro M, Salazar E, Marple K, Gupta G (2018) Constraint answer set programming without grounding. Theory and Practice of Logic Programming 18(3–4):337–354
Calcagno C, Distefano D, O'Hearn PW, Yang H (2011) Compositional shape analysis by means of bi-abduction. J. ACM 58(6)
Caroprese L, Trubitsyna I, Truszczyński M, Zumpano E (2014) A measure of arbitrariness in abductive explanations. Theory and Practice of Logic Programming 14(4–5):665–679
Caroprese L, Zumpano E, Bogaerts B (2022) Computing abductive explanations. IEEE Intell. Syst. 37(6):18–26
Chen Y, Su Z, (2015) Guided differential testing of certificate validation in SSL, TLS implementations. ESEC, FSE, (2015) Association for Computing Machinery. NY, USA, New York
Chen Y, Su T, Sun C, Su Z, Zhao J (2016) Coverage-directed differential testing of JVM implementations. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'2016. Association for Computing Machinery, New York, NY, USA, pp 85–99
Christiansen H, Dahl V (2005) HYPROLOG: A new logic programming language with assumptions and abduction. In: Gabbrielli M, Gupta G (eds) Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2–5, 2005, Proceedings. Lecture Notes in Computer Science, vol 3668, pp 159–173
Douven I (2021) Abduction. In: Zalta EN (ed) The Stanford Encyclopedia of Philosophy, Summer 2021 edition
Estévez-Martín S, Fernández AJ, Sáenz-Pérez F, Hortalá-González T, Rodríguez-Artalejo M, Vado Vírseda R (2009) On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in $CFLP$. Theory and Practice of Logic Programming 9(4):415–527
Frühwirth T (1998) Theory and practice of constraint handling rules. The Journal of Logic Programming 37(1):95–138
Fung TH, Kowalski R (1997) The IFF proof procedure for abductive logic programming. Journal of Logic Programming 33(2):151–165. https://doi.org/10.1016/S0743-1066(97)00026-5
Gavanelli M, Lamma E, Mello P, Milano M (2003) Torroni P (2003) Interpreting abduction in CLP. In: Buccafurri F (ed) 2003 Joint Conference on Declarative Programming. AGP-2003, Reggio Calabria, Italy, September 3–5, pp 25–35
Gavanelli M, Julián-Iranzo P, Sáenz-Pérez F (2024) PIFFC and PTS Systems. https://www.fdi.ucm.es/profesor/fernan/ALP/PIFFCv3.zip
Julián-Iranzo P, Sáenz-Pérez F (2021) Planning for an efficient implementation of hypothetical Bousi∼Prolog. Theory and Practice of Logic Programming 21(5):680–697
Kakas AC, Kowalski RA, Toni F (1992) Abductive logic programming. Journal of Logic and Computation 2:719–770. https://doi.org/10.1093/logcom/2.6.719
Kunen K (1987) Negation in logic programming. Journal of Logic Programming 4:289–308
Le V, Afshari M, Su Z (2014) Compiler validation via equivalence modulo inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI'2014. Association for Computing Machinery, New York, NY, USA, pp 216–226
McKeeman WM (1998) Differential testing for software. Digit. Tech. J. 10(1):100–107
Peirce CS (1965) Collected Papers of Charles Sanders Peirce [CP]. Editors Ch. Hartshorne and P. Weiss
Petsios T, Tang A, Stolfo S, Keromytis AD, Jana S (2017) Nezha: Efficient domain-independent differential testing. In: 38th IEEE Symposium on Security and Privacy (SP), pp 615–632. https://doi.org/10.1109/SP.2017.27
Wielemaker J (2022) SWI-Prolog. Backtrackable assert. https://www.swi-prolog.org/pldoc/man?predicate=undo/1
Wielemaker J, Schrijvers T, Triska M, Lager T (2012) SWI-Prolog. Theory and Practice of Logic Programming 12(1–2):67–96
Yang X, Chen Y, Eide E, Regehr J, (2011) Finding and Understanding Bugs in C Compilers. ACM SIGPLAN PLDI, (2011) PLDI'2011. ACM, New York, NY, USA, pp 283–294