Abduction with Hypotheses Confirmation

Marco Alberti[1] Marco Gavanelli[1], Evelina Lamma[1], Paola Mello[2], and Paolo Torroni[2]

[1] DI - University of Ferrara - Via Saragat, 1 - 44100 Ferrara, Italy.
{malberti|mgavanelli|elamma}@ing.unife.it

[2] DEIS - University of Bologna - Viale Risorgimento, 2 - 40136 Bologna, Italy.
{pmello|ptorroni}@deis.unibo.it

Abstract. Abduction can be seen as the formal inference corresponding to human hypothesis making. It typically has the purpose of explaining some given observation. In classical abduction, hypotheses could be made on events that may have occurred in the past. In general, abductive reasoning can be used to generate hypotheses about events possibly occurring in the future (forecasting), or may suggest further investigations that will confirm or disconfirm the hypotheses made in a previous step (as in scientific reasoning). We propose an operational framework based on Abductive Logic Programming, extending existing frameworks in many respects, including accommodating dynamic observations and hypothesis confirmation.

1 Introduction

Often, reasoning paradigms in artificial intelligence mimic human reasoning, providing a formalization and a better understanding of the human basic inferences. Abductive reasoning can be seen as a formalization, in computational logics, of hypotheses making. In order to explain observations, we hypothesize that some (unknown) events have happened, or that some (not directly measurable) properties hold true. The hypothesized facts are then assumed as true, unless they are disconfirmed in the following.

Hypothesis making is particularly important in scientific reasoning: scientists will hypothesize properties about nature, which explain some observations; in subsequent work, they will try to prove (if possible), or at least to confirm the hypotheses. This process leads often to generating new alternative sets of hypotheses. Starting from hypotheses on the current situation, scientists try to foresee their possible consequences; this provides new hypotheses on the future behavior that will be confirmed or disconfirmed by the actual events.

A typical application of abductive reasoning is *diagnosis*. Starting from the observation of symptoms, physicians hypothesize in general possible alternative diseases that may have caused them. Following an iterative process, they will try to support their hypotheses, by prescribing further exams, of which they foresee the possible alternative results. They will then drop the hypotheses disconfirmed by such results, and take as most faithful those supported by them. New findings, such as results of exams or new symptoms, may help generating new hypotheses.

We can then describe this kind of hypothetical reasoning process as composed by three main elements: classically, *explaining observations*, by assuming possible causes of the observed effects; but also, *adapting* such assumptions to upcoming events, such as new symptoms occurring, and *foreseeing* the occurrence of new events, which may or may not occur indeed.

In Abductive Logic Programming, many formalisms have been proposed [1–6], along with proof procedures able to provide, given a knowledge base and some observation, possible sets of hypotheses that explain the observation. Integrity Constraints are used to drive the process of hypothesis generation, to make such sets consistent, and possibly to suggest new hypotheses. Most frameworks focus on one aspect of abductive reasoning: assumption making, based on a static knowledge and on some observation.

In this work, we extend the concepts of abduction and abductive proof procedures in two main directions. We cater for the dynamic acquisition of new facts (events), which possibly have an impact on the abductive reasoning process, and for confirmation (or disconfirmation) of hypotheses based on such events. We propose a language, able to state desired properties of the events supporting the hypotheses: for instance, we could say that, given some combination of hypotheses and facts, we make the hypothesis that some new events will occur. We call this kind of hypothesis *expectation*. Expectations can be "positive (to be confirmed by certain events occurring), or "negative" (to be confirmed by certain events not occurring). For this purpose, we express expectations as abducible literals.

In our framework, we need to be able to state that some event is expected to happen *within some time interval*: if the event does actually happen within it, the hypothesis is confirmed, it is disconfirmed otherwise. In doing so, we need to introduce *variables* (e.g. to model time), and to state *constraints* on variables occurring in abducible atoms. Moreover, possible expectation could be involving universal quantification: this typically happens with negative expectations ("The patient is expected *not* to show symptom Q *at all times*"). For this reason, we also need to cater for abducibles possibly containing universally quantified variables.

To summarize, the main new features of the present work with respect to classical ALP frameworks are:

– dynamic update of the knowledge base to cater for new events, whose occurrence interacts with the abductive reasoning process itself;
– confirmation and disconfirmation of hypotheses, by matching expectations and actual events;
– hypotheses with universally quantified variables;
– constraints à la Constraint Logic Programming [7].

We do it by defining syntax, declarative and operational semantics of an abductive framework, based on an extension of the IFF proof procedure [4], called $\mathcal{S}$CIFF[8].[3] The $\mathcal{S}$CIFF has been implemented using *Constraint Handling Rules* [9]. Being the $\mathcal{S}$CIFF an extension of an existing abductive framework, it

[3] Historically, the name $\mathcal{S}$CIFF is due to the fact that this framework has been firstly applied to modelling protocol in agent $\mathcal{S}$ocieties, and that is also deals with CLP $\mathcal{C}$onstraints.

also caters for classic abductive logic programming (static knowledge, no notion of confirmation by events). However, due to space limitations, in this work we only focus on the original new parts.

In the following Sect. 2 we introduce our framework's knowledge representation. In Sect. 3 and 4 we provide declarative and operational semantics, and we show a soundness result. In Sect. 5 we give some information about its current implementation. In Sect. 6 we show an example of the functioning of the SCIFF in a multi-agent setting. Before concluding, we discuss about related work in Sect. 7. Additional details about the syntax of data structures used by the SCIFF and allowedness criteria used to prove soundness are given in appendix.

2 Knowledge Representation

In this section we show the knowledge representation of the abstract abductive framework of the SCIFF. The knowledge base dynamically evolves as new events are known. It is represented by the 5-tuple $\langle KB, \mathbf{HAP}, \mathbf{EXP}, \mathcal{IC}_S, \mathcal{G}\rangle$, where:

- KB is the knowledge base (an extended logic program);
- $\mathbf{HAP}$ is the *History* of *happened events*: atoms indicated with functor $\mathbf{H}$;
- $\mathbf{EXP}$ is the set of abduced *expectations*: literals indicated by the functors $\mathbf{E}$, $\mathbf{EN}$, $\neg\mathbf{E}$ and $\neg\mathbf{EN}$;
- $\mathcal{IC}_S$ is the set of *Integrity Constraints* (IC$_S$); and
- $\mathcal{G}$ is the set of *Goals*.

An instance of the abductive system (in the following, we will call an abductive framework ALP, and $ALP_{\mathbf{HAP}}$ a specific instance of it) takes into account occurred events ($\mathbf{HAP}$). Events are taken from an event queue, which is not modelled here. *Expectations* can represent (*i*) events that should (but might not) happen (and they are represented as atoms indicated with functor $\mathbf{E}$), or (*ii*) events that should not (but might indeed) happen (and they are represented as atoms indicated with functor $\mathbf{EN}$), in order for the previous hypotheses to be confirmed. Their (default) negation is written as $\neg\mathbf{E}/\neg\mathbf{EN}$.

The full syntax of our language is reported in Appendix A. We conclude this section with a simple example in the medical domain, where abduction is used to diagnose diseases starting from symptom observation. The aim of this example is to show the two main improvements of the SCIFF with respect to previous work: the dynamic acquisition of new facts, and the confirmation of hypotheses by events.

Let us consider a symptom s, which can be explained by abducing one of three types of diseases, of which the first and the third are incompatible, and the second is accompanied by a condition (the patient's temperature is expected to increase):

$$
\begin{array}{l}
symptom(s) :- \mathbf{E}(disease(d_1)), \mathbf{EN}(disease(d_3)).\\
symptom(s) :- \mathbf{E}(disease(d_2)), \mathbf{E}(temperature(high)).\\
symptom(s) :- \mathbf{E}(disease(d_3)), \mathbf{EN}(disease(d_1)).
\end{array}
$$

$\mathcal{IC}_S$ expresses what is expected or *should* happen or not, given some happened events and/or some abduced hypotheses. They are in the form of implications, and can involve both literals defined in the KB, and expectations and events in **EXP**and **HAP**. For example, an IC_S in $\mathcal{IC}_S$ could state that if the result of some exam r is positive, then we can hypothesize that the patient is not affected by disease d_1:

$$\mathbf{H}(result(r, positive)) \rightarrow \mathbf{EN}(disease(d_1))$$

Abducing $\mathbf{EN}(disease(d_1))$ would rule out, in our framework, the possibility to abduce $\mathbf{E}(disease(d_1))$. We see how the dynamic occurrence of new events can drive the generation and selection of abductive explanations of goals. Let us now assume that the patient, at some point, shows the symptom $temperature(low)$. The following constraint can be used to express this fact to be inconsistent with an expectation about his temperature increasing:

$$\mathbf{E}(temperature(high)) \rightarrow \mathbf{EN}(temperature(low))$$

If the diagnosis $\mathbf{E}(disease(d_2)), \mathbf{E}(temperature(high))$ is chosen for s, this IC_S would have as a consequence the generation of the expectation $\mathbf{EN}(temperature(low))$, which would be frustrated by the fact $\mathbf{H}(temperature(low))$. The only possible explanation for s thus remains $\mathbf{E}(disease(d_3)), \mathbf{EN}(disease(d_1))$. We see by this example how the hypotheses can be disconfirmed by events.

The abductive system will usually have a goal, which typically is an observation for which we are searching for explanations; for example, a conjunction of *symptom* atoms.

3 Declarative semantics

In the previous section, we have defined an instance of the abductive framework as a tuple $\langle KB, \mathbf{HAP}, \mathbf{EXP}, \mathcal{IC}_S, \mathcal{G}\rangle$. In this section, we propose an abductive interpretation for $ALP_{\mathbf{HAP}}$, depending on the events in the history **HAP**. We adopt a three-valued logic, where literals of kind $\mathbf{H}()$ or $\neg\mathbf{H}()$ can be interpreted as true, false or unknown.

Throughout this section, for the sake of simplicity, we always consider the ground version of the knowledge base and integrity constraints, and do not consider CLP-like constraints.

The set of expectations that we want to generate should satisfy the properties that we list in the following. Firstly, we are interested in sets of expectations that are compatible with the KB and the set **HAP**, and with $\mathcal{IC}_S$.

Definition 1. **$\mathcal{IC}_S$-consistency.** *Given an instance $ALP_{\mathbf{HAP}}$, an $\mathcal{IC}_S$-*consistent *set of expectations* **EXP** *is a set of expectations such that:*

$$Comp(KB \cup \mathbf{EXP}) \cup \mathbf{HAP} \cup CET \models \mathcal{IC}_S \tag{1}$$

where Comp is three-valued completion [10] and CET Clark's equational theory.

$\mathcal{IC}_S$-consistent sets of expectations can be however self-contradictory (e.g., both $\mathbf{E}(p)$ and $\neg\mathbf{E}(p)$ may belong to a $\mathcal{IC}_S$-consistent set). Therefore, we define two other classes of consistency: E-consistency and $\neg$-consistency.

Definition 2. *A set of expectations* $\mathbf{EXP}$ *is* E-consistent *if and only if for each (ground) term p:* $\{\mathbf{E}(p), \mathbf{EN}(p)\} \not\subseteq \mathbf{EXP}$

A set of expectations $\mathbf{EXP}$ *is* $\neg$-consistent *if and only if for each (ground) term p:* $\{\mathbf{E}(p), \neg\mathbf{E}(p)\} \not\subseteq \mathbf{EXP}$ *and* $\{\mathbf{EN}(p), \neg\mathbf{EN}(p)\} \not\subseteq \mathbf{EXP}$.[4]

Given an instance of an ALP, we name *admissible* a set of expectations which satisfies Definitions 1 (Eq. 1), and 2, i.e. which is $\mathcal{IC}_S$-, E- and $\neg$-consistent.

Definition 3. Confirmation. *Given an instance $ALP_{\mathbf{HAP}}$, a set of expectations* $\mathbf{EXP}$ *is* confirmed *if and only if for each (ground) term p:*

$$\mathbf{HAP} \cup Comp(\mathbf{EXP}) \cup \{\mathbf{E}(p) \rightarrow \mathbf{H}(p)\} \cup \{\mathbf{EN}(p) \rightarrow \neg\mathbf{H}(p)\} \cup CET \not\models \bot \quad (2)$$

If Eq. 2 does not hold, the set of expectations is called disconfirmed.

Note that we keep the same completion semantics (with the CET) taken by the IFF proof procedure. However, we do not complete the set $\mathbf{HAP}$, as new events may occur in the following.

Definition 3 requires that each negative expectation in $\mathbf{EXP}$ has no corresponding happened event, while it is weaker for positive expectations. In fact, in general, we cannot disconfirm positive expectations, unless we assume at some point that no more events will happen. In that case, a positive expectation can be disconfirmed for instance if some deadlines are missed. To this purpose, we introduce the following assumption:

Definition 4. Full temporal knowledge. *We suppose that all the (significant) events that have happened are known to the abductive system at any time.*

Finally, an instance of an abductive framework should explain the given observations:

Definition 5. Goal provability. *Given an instance $ALP_{\mathbf{HAP}}$ and a goal G, we say that G is* provable *(and we write $ALP_{\mathbf{HAP}} \approx_{\mathbf{EXP}} G$) iff there exists an admissible and confirmed set of expectations* $\mathbf{EXP}$*, such that:*

$$Comp(KB \cup \mathbf{EXP}) \cup \mathbf{HAP} \cup CET \models G \quad (3)$$

We conclude this section with a final remark about the relation of this declarative semantics with that of other abductive systems of literature. Usually, an Abductive Logic Program is defined to be a triple $ALP = \langle KB, \mathcal{A}, \mathcal{IC}\rangle$, where $\mathcal{A}$ is a set of abducible predicates, KB is a logic program and $\mathcal{IC}$ a set of integrity constraints [11]. The abductive explanation of a goal/observation g given an ALP is a set of hypotheses $\Delta \subseteq \mathcal{A}$. If we consider a given static history of events, and we map $\mathbf{EXP}$ onto such a Δ, Eq. 1 and Eq. 3 correspond to the equations defining the declarative semantics of a classic ALP framework [4]. Def. 2 and 3 clearly show our extensions with respect to existing approaches.

[4] For abducibles, we adopt the same viewpoint as in ACLP [5]: for each abducible predicate A, we have also the abducible predicate $\neg A$ for the negation of A together with the integrity constraint $(\forall X)\neg A(X), A(X) \rightarrow \bot$.

4 Operational Semantics

Our framework's IC_S are very much related to the integrity constraints of the IFF proof procedure [4]. This leads to the idea of using an extension of the IFF proof procedure for generating expectations, and check for their confirmation.

In particular, the additional features that we need are the following: (*i*) accept new events as they happen, (*ii*) produce a (disjunction of) set of expectations, (*iii*) detect confirmation of expectations, (*iv*) detect disconfirmation as soon as possible.

The proof procedure that we are about to present is called SCIFF. Following Fung and Kowalski's approach [4], we describe the SCIFF as a transition system. Due to space limitations, we will only focus here on the new transitions, while the reader can refer to [4] for the basic IFF transitions.

4.1 Data Structures

The SCIFF proof procedure is based on a transition system. Each state is defined by the tuple $T \equiv \langle R, CS, PSIC, \mathbf{EXP}, \mathbf{HAP}, \mathbf{CONF}, \mathbf{DISC} \rangle$, where R is the resolvent, CS is the constraint store, $PSIC$ is the set of partially solved integrity constraints, **EXP** is the set of (pending) expectations, **HAP** is the history of happened events, **CONF** is a set of confirmed hypotheses, **DISC** is a set of disconfirmed expectations.

Variable quantification In the IFF proof procedure, all the variables that occur in the resolvent or in abduced literals are existentially quantified, while the others (appearing only in implications) are universally quantified. Our proof procedure has to deal with universally quantified variables in the abducibles and in the resolvent. In the IFF proof procedure, variables in an implication are existentially quantified if they also appear in an abducible or in the resolvent. In our language, we can have existentially quantified variables in the integrity constraints even if they do not occur elsewhere (see Appendix A.3).

For all these reasons, in the operational semantic specification we leave the variable quantification explicit. Moreover, we need to distinguish between variables that appear in abduced literals (or in the resolvent) and variables occurring only in integrity constraints. The scope of the variables in abduced literals and in the resolvent is the whole tuple. The scope of the other variables is the implication in which they occur.

Initial Node and Success A derivation D is a sequence of nodes

$$T_0 \to T_1 \to \ldots \to T_{n-1} \to T_n.$$

Given a goal G and a set of integrity constraints $\mathcal{IC}_S$, the first node is:

$$T_0 \equiv \langle \{G\}, \emptyset, \mathcal{IC}_S, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

i.e., the resolvent is initially the query ($R_0 = \{G\}$) and the partially solved integrity constraints $PSIC$ is the set of integrity constraints ($PSIC_0 = \mathcal{IC}_S$).

The other nodes $T_j, j > 0$, are obtained by applying the transitions defined in the next section, until no transition can be applied anymore (quiescence). Every arc in a derivation is labelled with the name of a transition.

Definition 6. *Starting with an instance* $ALP_{\mathbf{HAP}^i}$ *there exists a* successful derivation *for a goal* G *iff the proof tree with root node* $\langle\{G\}, \emptyset, \mathcal{IC}_S, \emptyset, \mathbf{HAP}^i, \emptyset, \emptyset\rangle$ *has at least one leaf node* $\langle\emptyset, CS, PSIC, \mathbf{EXP}, \mathbf{HAP}^f, \mathbf{CONF}, \emptyset\rangle$ *where* $\mathbf{HAP}^f \supseteq \mathbf{HAP}^i$ *and* CS *is consistent (i.e., there exists a ground variable assignment such that all the constraints are satisfied). In that case, we write:*

$$\mathcal{S}_{\mathbf{HAP}^i} \vdash^{\mathbf{HAP}^f}_{\mathbf{EXP} \cup \mathbf{CONF}} G$$

From a non-failure leaf node N, answers can be extracted in a very similar way to the IFF proof procedure. First, a substitution σ' is computed such that σ' replaces all variables in N that are not universally quantified by ground terms, and σ' satisfies all the constraints in the store CS_N. If the constraint solver is (theory) complete (i.e., for each set of constraints c, the solver always returns *true* or *false* [12], and never *unknown*), then there will always exist a substitution σ' for each success node N. Otherwise, if the solver is incomplete, σ' may not exist: this is discovered during the answer extraction phase. In such a case, the node N will be marked as failure node, and another leaf node can be selected (if there exists one).

Definition 7. *Let* $\sigma = \sigma'|_{vars(G)}$ *be the restriction of* σ' *to the variables occurring in the initial goal* G. *Let* $\Delta = (\mathbf{CONF}_N \cup \mathbf{EXP}_N)/\sigma'$. *The pair* (Δ, σ) *is the* abductive answer *obtained from the node* N.

The SCIFF proof procedure performs some inferences based on the semantics of time, under the full temporal knowledge assumption (Def. 4).

4.2 Transitions

The transitions are those of the IFF proof procedure, enlarged with those of CLP [7], and with specific transitions accommodating the concepts of confirmation of hypotheses, dynamically growing history and consistency of the set of expectations with respect to the given definitions (Definitions 1 and 2).

In this section, the letter k will indicate the level of a node; each transition will generate one or more nodes from level k to $k + 1$. We will not explicitly report the new state for items that do not change; e.g., if a transition generates a new node from the node

$$T_k \equiv \langle R_k, CS_k, PSIC_k, \mathbf{EXP}_k, \mathbf{HAP}_k, \mathbf{CONF}_k, \mathbf{DISC}_k\rangle$$

and we do not explicitly state the value of R_{k+1}, it means that $R_{k+1} = R_k$.

IFF-like transitions The SCIFF proof procedure contains transitions borrowed from the IFF proof procedure, namely *Unfolding*, *Propagation*, *Splitting*, *Case Analysis*, *Factoring*, *Equivalence Rewriting* and *Logical Equivalence*. They have been extended to cope with abducibles containing universally quantified variables and with CLP constraints. We omit them here for lack of space; the basic transitions were proposed by Fung and Kowalski [4], while the extended ones can be found in a technical report [13].

Dynamically growing history The happening of events is considered by a transition *Happening*. This transition takes an event $\mathbf{H}(Event)$ from the external queue and puts it in the history $\mathbf{HAP}$; the transition *Happening* is applicable only if an *Event* such that $\mathbf{H}(Event) \notin \mathbf{HAP}$ is in the external queue.

Formally, from a node N_k transition *Happening* produces a single successor

$$\mathbf{HAP}_{k+1} = \mathbf{HAP}_k \cup \{\mathbf{H}(Event)\}.$$

Note that transition *Happening* should be applied to all the non-failure nodes (in the frontier).

Confirmation and Disconfirmation

Disconfirmation **EN** Given a node N with the following situation:

$$\mathbf{EXP}_k = \mathbf{EXP'} \cup \{\mathbf{EN}(E_1)\} \qquad \mathbf{HAP}_k = \mathbf{HAP'} \cup \{\mathbf{H}(E_2)\}$$

Disconfirmation **EN** produces two nodes N^1 and N^2, as follows:

N^1			N^2		
$\mathbf{EXP}^1_{k+1}$	$=$	$\mathbf{EXP'}$	$\mathbf{EXP}^2_{k+1}$	$=$	$\mathbf{EXP}_k$
$\mathbf{DISC}^1_{k+1}$	$=$	$\mathbf{DISC}_k \cup \{\mathbf{EN}(E_1)\}$	$\mathbf{DISC}^2_{k+1}$	$=$	$\mathbf{DISC}_k$
CS^1_{k+1}	$=$	$CS_k \cup \{E_1 = E_2\}$	CS^2_{k+1}	$=$	$CS_k \cup \{E_1 \neq E_2\}$

Example 1. Suppose that $\mathbf{HAP}_k = \{\mathbf{H}(p(1,2))\}$ and $\exists X \forall Y \mathbf{EXP}_k = \{\mathbf{EN}(p(X,Y))\}$. *Disconfirmation* **EN** will produce the two following nodes:

$\exists X \forall Y \mathbf{EXP}_k = \{\mathbf{EN}(p(X,Y))\}$ $\mathbf{HAP}_k = \{\mathbf{H}(p(1,2))\}$

$CS^1_{k+1} = \{X = 1 \wedge Y = 2\}$ $CS^2_{k+1} = \{X \neq 1 \vee Y \neq 2\}$
$\mathbf{DISC}^1_{k+1} = \{\mathbf{EN}(p(1,2))\}$
$CS_{k+2} = \{X \neq 1\}$

where the last simplification in the right branch is due to the rules of the constraint solver (see Section CLP).

Confirmation **E** Starting from a node N as follows:

$$\mathbf{EXP}_k = \mathbf{EXP'} \cup \{\mathbf{E}(E_1)\}, \qquad \mathbf{HAP}_k = \mathbf{HAP'} \cup \{\mathbf{H}(E_2)\}$$

Confirmation **E** builds two nodes, N^1 and N^2; in node N^1 we assume that the expectation and the happened event unify, and in node N^2 we hypothesize that the two do not unify:

N^1			N^2		
$\mathbf{EXP}^1_{k+1}$	$=$	$\mathbf{EXP'}$	$\mathbf{EXP}^2_{k+1}$	$=$	$\mathbf{EXP}_k$
$\mathbf{CONF}^1_{k+1}$	$=$	$\mathbf{CONF}_k \cup \{\mathbf{E}(E_1)\}$	$\mathbf{CONF}^2_{k+1}$	$=$	$\mathbf{CONF}_k$
CS^1_{k+1}	$=$	$CS_k \cup \{E_1 = E_2\}$	CS^2_{k+1}	$=$	$CS_k \cup \{E_1 \neq E_2\}$

Disconfirmation **E** In order to check the disconfirmation of a positive expectation **E**, we need to assume that there will never be a matching event in the external queue. We can, e.g., exploit the semantics of *time*. If we make the hypothesis of full temporal knowledge (Definition 4), we can infer that an expected event for which the deadline is passed, is disconfirmed.

Given a node:

- $\mathbf{EXP}_k = \{\mathbf{E}(X,T)\} \cup \mathbf{EXP}'$
- $\mathbf{HAP}_k = \{\mathbf{H}(Y,T_c)\} \cup \mathbf{HAP}'$
- $\forall E_2, T_2 : \mathbf{H}(E_2,T_2) \in \mathbf{HAP}_k$, $CS_k \cup \{(E_2,T_2) = (X,T)\} \models \bot$
- $CS_k \models T < T_c$

transition Disconfirmation **E** is applicable and creates the following node:

- $\mathbf{EXP}_{k+1} = \mathbf{EXP}'$
- $\mathbf{DISC}_{k+1} = \mathbf{DISC}_k \cup \{\mathbf{E}(X,T)\}$.

Operationally, one can often avoid checking that (X,T) does not unify with every event in the history by choosing a preferred order of application of the transitions. By applying Disconfirmation **E** only if no other transition is applicable, the check can be safely avoided.

Notice that this transition infers the current time from happened event; i.e., it infers that the current time cannot be less than the time of a happened event.

Symmetrically to Disconfirmation **E**, we also have a transition *Confirmation* **EN**, which we do not report here for lack of space; the interested reader is referred to [13].

Note that the entailment of constraints from a constraint store is, in general, not easy to verify. In the particular case of $CS_k \models T < T_c$, however, we have that the constraint $T < T_c$ is unary (T_c is always ground), thus a CLP for finite domains solver CLP(FD) is able to verify the entailment very easily if the store contains only unary constraints (it is enough to check the maximum value in the domain of T). Moreover, even if the store contains non-unary constraints (thus the solver performs, in general, incomplete propagation), the transition will not compromise the soundness and completeness of the proof procedure. If the solver performs a powerful propagation (including pruning, in CLP(FD)), the disconfirmation will be early detected, otherwise, it will be detected later on.

Consistency In order to ensure **E**-consistency of the set of expectations, we impose the following integrity constraint:

$$\mathbf{E}(T) \wedge \mathbf{EN}(T) \rightarrow \bot$$

while for $\neg$-consistency, we impose the following integrity constraints:

$$\mathbf{E}(T) \wedge \neg\mathbf{E}(T) \rightarrow \bot \qquad \mathbf{EN}(T) \wedge \neg\mathbf{EN}(T) \rightarrow \bot$$

CLP Here we adopt the same transitions as in CLP [7]. Therefore, the symbols $=$ and $\neq$ are in the constraint language. Note that a constraint solver works on a constraint domain which has an associated interpretation. In addition, the constraint solver should handle the constraints among terms derived from the unification. Therefore, beside the specific constraint propagation on the constraint domain, we need further inference rules for coping with the unification. For space limitations, we cannot show them here: but they can be found in Appendix A.4.

For instance, CLP transitions avoid having a node with such an expectation as $\mathbf{E}(p(X))$ with the $X > 3 \wedge X < 2$ CLP constraints on X.

4.3 Soundness

The following proposition relates the operational notion of successful derivation with the corresponding declarative notion of goal provability.

Proposition 1. *Given an instance $ALP_{\mathbf{HAP}^i}$ of an ALP program and a ground goal G, if $ALP_{\mathbf{HAP}^i} \mid\!\sim^{\mathbf{HAP}}_{\mathbf{EXP} \cup FULF} G$ then $ALP_{\mathbf{HAP}} \mid\!\approx_{\mathbf{EXP} \cup FULF} G$.*

This property has been proven for some notable classes of ALP programs. In particular, a proof of soundness can be found in [13] for *allowed* ALPs (for a definition of allowedness see Section A.3). The proof is based on a correspondence drawn between the SCIFF and IFF transitions, and exploits the soundness results of the IFF proof procedure [4].

5 Implementation

The SCIFF proof procedure, defined in Section 4, has been implemented using *Constraint Handling Rules* (*CHR*). *CHR* [9] are essentially a committed-choice language consisting of guarded rules that rewrite constraints in a store into simpler ones until they are solved. *CHR* define both *simplification* (replacing constraints by simpler constraints while preserving logical equivalence) and *propagation* (adding new, logically redundant but computationally useful, constraints) over user-defined constraints.

The main intended use for *CHR* is to write constraint solvers, or to extend existing ones. However, although ours is not a classic constraint programming setting, the computational model of *CHR* presents features that make it a useful tool for the implementation of the SCIFF proof procedure.

Following Fung and Kowalski's approach [4], we defined the proof tee independent of the search strategy, which has to be defined instead at the implementation level. The current implementation is based on a depth-first strategy [8]. This choice, enabling us to tailor the implementation for the built-in computational features of Prolog, allows for a both simpler and more efficient implementation of the proof procedure, although, in this way, we miss the possibility of a global representation of the *frontier* of the proof tree.

The implementation is based on the following idea:

– the data structures of the proof procedure (Section 4.1) are mapped into CHR constraints

– the transitions are mapped into CHR rules.

In a sense, the data structures are seen as new constraints whose propagation is defined by the transitions of the proof procedure.

Moreover, variables are represented by attributed variables [14]. Attributes can represent a variable's quantification and flagging, as well as quantifier restrictions: all of these attributes influence a variable's behavior with respect to unification.

For unification and non-unification constraints, reified unification is exploited: the ternary *CHR* constraint `reif_unify(Term1,Term2,B)` expresses that `Term1` and `Term2` unify iff `B` $= 1$ (`B` is a binary variable). An *ad hoc* constraint solver has been implemented to handle reified unification, which takes into account variable attributes such as quantification. A description of the implemented system can be found in [15].

6 Using the *S*CIFF for agent compliance verification

Abduction has been used for various applications, and many of them (e.g. diagnosis) could benefit from an extension featuring hypotheses confirmation, such as the one depicted in this paper. We have applied the language to a multi-agent setting, in the context of the SOCS project [16].

In order to combine autonomous agents and have them operate in a coordinated fashion, protocols are often defined. Protocols show, in a way, the ideal behaviour of agents. But, since agents are often assumed to be autonomous, and societies open and heterogeneous, agent compliance to protocols is rarely a reasonable assumption to make. Agents may violate the protocols due to malicious intentions, to wrong design or, for instance, to failure to keep the pace with tight deadlines.

With our language, protocols can be easily formalized, and the *S*CIFF proof procedure can be used then to check whether the agents comply to protocols. For instance, let us consider the (very simple) *Query-ref* protocol: an agent requests a piece of information to another agent, which may either provide it or refuse it, but not both. The protocol can be expressed by the following three integrity constraints (where D represents a dialogue identifier):

$\mathbf{IC_1}$: $\mathbf{H}(tell(A, B, query\text{-}ref(Info), D), T) \rightarrow$
$\quad\mathbf{E}(tell(B, A, inform(Info, Answer), D), T_1) \wedge T_1 \leq T + 10 \vee$
$\quad\mathbf{E}(tell(B, A, refuse(Info), D), T_1) \wedge T_1 \leq T + 10$

$\mathbf{IC_2}$: $\mathbf{H}(tell(A, B, inform(Info, Answer), D), T) \rightarrow$
$\quad\mathbf{EN}(tell(A, B, refuse(Info), D), T_1) \wedge T_1 \geq T$

$\mathbf{IC_3}$: $\mathbf{H}(tell(A, B, refuse(Info), D), T) \rightarrow$
$\quad\mathbf{EN}(tell(A, B, inform(Info, Answer), D), T_1) \wedge T_1 \geq T$

IC_1 expresses that an agent that receives a *query-ref* must reply with either an *inform* or a *refuse* by 10 time units; IC_2 and IC_3 state that an agent that performs an *inform* cannot perform a *refuse* later, and *vice-versa.*

Let us suppose that the following events happen:

$\mathbf{H_1}$: $\mathbf{H}(tell(alice, bob, query\text{-}ref(what\text{-}time), d_0), 10)$

$\mathbf{H_2}$: $\mathbf{H}(tell(bob, alice, refuse(what\text{-}time), d_0), 15)$

and consider how SCIFF verifies that such an history is compliant to the interaction protocol.

The $\mathbf{H_1}$ event, by propagation of $\mathbf{IC_1}$, will cause a disjunction of two expectations to be generated, which will split the proof tree into two branches:

1. In the first branch, $\mathbf{EXP} = \{\mathbf{E}(\textit{tell}(\textit{bob}, \textit{alice}, \textit{inform}(\textit{what-time}, \textit{Answer}), d_0), T_1)\}$ and $\mathbf{CS} = \{T_1 \leq 20\}$. When $\mathbf{H_2}$ happens, by propagation of $\mathbf{IC_2}$, $\mathbf{EXP} = \{\mathbf{E}(\textit{tell}(\textit{bob}, \textit{alice}, \textit{inform}(\textit{what-time}, \textit{Answer}), d_0), T_1),$
$\mathbf{EN}(\textit{tell}(\textit{bob}, \textit{alice}, \textit{inform}(\textit{what-time}, \textit{Answer}), d_0), T_2)\}$, with $\mathbf{CS} = \{T_1 \leq 20, T_2 \geq 15\}$. Then, by enforcing E-consistency, the domain of T_1 is reduced: $\mathbf{CS} = \{T_1 \leq 14, T_2 \geq 15\}$. Now, *Disconfirmation*-$\mathbf{E}$ can be applied:
$\mathbf{E}(\textit{tell}(\textit{bob}, \textit{alice}, \textit{inform}(\textit{what-time}, \textit{Answer}), d_0), T_1)$ is moved from **EXP** to **DISC**; this means that this branch cannot be successful.
2. In the second branch, $\mathbf{EXP} = \{\mathbf{E}(\textit{tell}(\textit{bob}, \textit{alice}, \textit{refuse}(\textit{what-time}), d_0), T_1)\}$ and $\mathbf{CS} = \{T_1 \leq 20\}$. By propagation of $\mathbf{IC_2}$, after $\mathbf{H_2}$,
$\mathbf{EXP} = \{\mathbf{E}(\textit{tell}(\textit{bob}, \textit{alice}, \textit{refuse}(\textit{what-time}), d_0), T_1),$
$\mathbf{EN}(\textit{tell}(\textit{bob}, \textit{alice}, \textit{inform}(\textit{what-time}, \textit{Answer}), d_0), T_2)\}$, with $\mathbf{CS} = \{T_1 \leq 20, T_2 \geq 15\}$. By *Confirmation* **E**, $\mathbf{H_2}$ also causes
$\mathbf{E}(\textit{tell}(\textit{bob}, \textit{alice}, \textit{refuse}(\textit{what-time}), d_0), T_1)$ to be moved from **EXP** to **CONF**, with $T_1 = 15$. If no more events happen, this branch is successful.

Through this simple example, we showed how a protocol can be easily cast in our model. More rules can be easily added, to accomplish more complex protocols. In other work, we have shown the application of this formalism to a range of protocols [17, 18]. The use of expectations generated by the SCIFF could be manifold: by associating Confirmation/Disconfirmation with a notion of Fulfillment/Violation, we can verify at run-time the compliance of agents to protocols. Moreover, expectations, if made public, could be used by agents planning their activities, helping their choices if they aim at exhibiting a compliant behaviour.

7 Related Work

This work is mostly related to the IFF proof procedure [4], which it extends in several directions, as stated in the introduction.

Other proof procedures deal with constraints; in particular we mention ACLP [5] and the $\mathcal{A}$-system [6], which are deeply focussed on efficiency issues. Both of these proof procedures use integrity constraints in the form of denials (e.g., $A, B, C \rightarrow \bot$), instead of forward rules as the IFF (and SCIFF). Both of these proof procedures only abduce existentially quantified atoms, and do not consider quantifier restrictions, which make the SCIFF in this sense more expressive.

Some conspicuous work has been done with the integration of the IFF proof procedure with constraints [19]; however the integration is more focussed on a theoretical uniform view of abducibles and constraints than to an implementation of a proof procedure with constraints.

In [20], Endriss et al. present an implementation of an abductive proof procedure that extends IFF [4] in two ways: by dealing with constraint predicates and with non-allowed abductive logic programs. The cited work, however, does not deal with confirmation and disconfirmation of hypotheses and universally quantified variables in abducibles (**EN**), as ours does. The two works also differ in their implementation: Endriss et al.'s is a metainterpreter which exploits a built-in constraint solver, whereas we implement the proof transitions and variable unification by means of CHR and attributed variables. Both works have been conducted in the context of the SOCS project [16]: the main application of Endriss et al.'s is the implementation of the internal agent reasoning, while ours is the compliance check of the observable (external) agent behaviour.

Many other abductive proof procedures have been proposed in the past; the interested reader can refer to the exhaustive survey by Kakas et al. [11].

In [22], Sergot proposed a general framework, called *query-the-user*, in which some of the predicates are labelled as "askable"; the truth of askable atoms can be asked to the user. The framework provides, thus, the possibility of gathering new information during the computation. Our **E** predicates may in a sense be seen as asking information, while **H** atoms may be considered as new information provided during search. However, as we have shown in the paper, **E** atoms may also mean *expected* behavior, and the $\mathcal{S}$CIFF can cope with abducibles containing universally quantified variables.

The concept of hypotheses confirmation has been studied also by Kakas and Evans [21], where hypotheses can be corroborated or refuted by matching them with observable atoms: an explanation fails to be corroborated if some of its logical consequences are not observed. The authors suggest that their framework could be extended to take into account dynamic events, eventually, queried to the user: *"this form of reasoning might benefit for the use of a query-the-user facility"*.

In a sense, our work can be considered as an extension of these works: it provides the concept of confirmation of hypotheses, as in corroboration, and it provides an operational semantics for dynamically incoming events. Moreover, we extend the work by imposing integrity constraints to better define the feasible combinations of hypotheses, and we let the program abduce non-ground atoms.

In Speculative Computation [23, 24] hypotheses are abduced and can be confirmed later on. It is a framework for a multi-agent setting with unreliable communication. When an agent asks a query to another agent, it also abduces its (default) answer; if the real answer arrives within a deadline, the hypothesis is confirmed or disconfirmed; otherwise the computation continues with the default. In our work, expectations can be confirmed by events, but the scope is different. In our work, if a deadline is missed the computation fails, as an hypothesis has been disconfirmed.

Other implementations have been given of abductive proof procedures in Constraint Handling Rules [25, 26]. Our implementation is more adherent to the theoretical operational semantics (in fact, every transition is mapped onto CHR rules) and exploits the uniform understanding of constraints and abducibles noted by Kowalski et al. [19].

Finally, in Section 6 we considered multi-agent systems to show an application of the $\mathcal{S}$CIFF. Some discussion about other formal approaches to protocol verification can be found in [13].

8 Conclusions

In this paper, we proposed an abductive logic programming framework which extends most previous work in several directions. The two main features of this framework are: the possibility to account for new dynamically upcoming facts, and the possibility to have hypotheses confirmed/disconfirmed by following observations and evidence. We proposed a language, and described its declarative and operational semantics. We implemented the proof-procedure for a system verifying the compliance of agents to protocols; the implementation can be downloaded from http://lia.deis.unibo.it/Research/sciff/ [8].

References

1. Poole, D.L.: A logical framework for default reasoning. Artificial Intelligence **36** (1988) 27–47
2. Kakas, A.C., Mancarella, P.: On the relation between Truth Maintenance and Abduction. In Fukumura, T., ed.: Proc. PRICAI-90, Nagoya, Japan, Ohmsha Ltd. (1990) 438–443
3. Console, L., Dupré, D.T., Torasso, P.: On the relationship between abduction and deduction. Journal of Logic and Computation **1** (1991) 661–690
4. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33** (1997) 151–165
5. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive Constraint Logic Programming. Journal of Logic Programming **44** (2000) 129–177
6. Kakas, A.C., van Nuffelen, B., Denecker, M.: A-System: Problem solving through abduction. In Nebel, B., ed.: Proc. 17th IJCAI, Seattle, WA, MKP (2001) 591–596
7. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming **19-20** (1994) 503–582
8. The SCIFF abductive proof procedure http://lia.deis.unibo.it/Research/sciff/.
9. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37** (1998) 95–138
10. Kunen, K.: Negation in logic programming. In: Journal of Logic Programming. Volume 4. (1987) 289–308
11. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In Gabbay, D.M., Hogger, C.J., Robinson, J.A., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 5., OUP (1998) 235–324
12. Jaffar, J., Maher, M., Marriott, K., Stuckey, P.: The semantics of constraint logic programs. Journal of Logic Programming **37(1-3)** (1998) 1–46
13. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of interaction protocols: a computational logic approach based on abduction. Tech. Rep. CS-2003-03, Dip. di Ingegneria, Ferrara, Italy (2003)
14. Holzbaur, C.: Specification of constraint based inference mechanism through extended unification. Dissertation, Dept. of Medical Cybernetics & AI, University of Vienna (1990)
15. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based tool. In Müller, J., Petta, P., eds.: Proc. (AT2AI-4 – EMCSR'2004 Session M), Vienna, Austria (2004)
16. Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees http://lia.deis.unibo.it/Research/SOCS/.

17. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Modeling interactions using *Social Integrity Constraints*: a resource sharing case study. In Leite, J.A., Omicini, A., Sterling, L., Torroni, P., eds.: Proc. DALT 2003. Melbourne, Victoria (2003) 81–96
18. Alberti, M., Daolio, D., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interaction protocols in a logic-based system. In: Proce. 19th SAC, AIMS Special Track, Nicosa, Cyprus, ACM Press (2004)
19. Kowalski, R., Toni, F., Wetzel, G.: Executing suspended logic programs. Fundamenta Informaticae **34** (1998) 203–224
20. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: Abductive Logic Programming with CIFF. In Bennett, B., ed.: Proc. 11th Workshop on Automated Reasoning, Leeds, UK, (2004) Extended Abstract.
21. Evans, C., Kakas, A.: Hypotheticodeductive reasoning. In: Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo (1992) 546–554
22. Sergot, M.J.: A query-the-user facility of logic programming. In Degano, P., Sandwell, E., eds.: Integrated Interactive Computer Systems, North Holland (1983) 27–41
23. Satoh, K., Inoue, K., Iwanuma, K., Sakama, C.: Speculative computation by abduction under incomplete communication environments. In: Proc. 4th ICMAS, Boston, USA, IEEE Press (2000) 263–270
24. Satoh, K., Yamamoto, K.: Speculative computation with multi-agent belief revision. In Castelfranchi, C., Lewis Johnson, W., eds.: Proc. AAMAS-2002, Part II, Bologna, Italy, ACM Press (2002) 897–904
25. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In Larsen, H., Kacprzyk, J., Zadrozny, S., Andreasen, T., Christiansen, H., eds.: FQAS, Flexible Query Answering Systems. LNCS, Warsaw, Poland, Springer (2000) 141–152
26. Gavanelli, M., Lamma, E., Mello, P., Milano, M., Torroni, P.: Interpreting abduction in CLP. In Buccafurri, F., ed.: APPIA-GULP-PRODE Joint Conf. on Declarative Programming, Reggio Calabria, Italy, (2003) 25–35
27. Lloyd, J.W.: Foundations of Logic Programming. 2nd extended edn. Springer-Verlag (1987)
28. Bürckert, H.: A resolution principle for constrained logics. Artificial Intelligence **66** (1994) 235–271

A Syntax

A.1 Syntax of the EXP

The sets **HAP** and **EXP** contain information about the (past) events and observations, and about the current expectations. There can be alternative **EXP**s for a given **HAP**, as usually there can be several explanations for a given event. The syntax of **HAP** and **EXP** is as follows:

$$\begin{aligned} \mathbf{HAP} &::= [\mathbf{H}(\mathit{Event}[,\, \mathit{Time}])]^\star \\ \mathbf{EXP} &::= \mathit{Expectation}\ [\ \wedge (\mathit{Expectation}|\mathit{Constraint})]^\star \\ \mathit{Expectation} &::= [\neg]\mathbf{E}(\mathit{Event}\ [,T])\ \mid\ [\neg]\mathbf{EN}(\mathit{Event}\ [,T]) \\ \mathit{Event} &::= \mathit{Term} \end{aligned} \tag{4}$$

Atom and *Term* are intended as usual in Logic Programming [27]. *Constraint* is a constraint in the CLP sense [7]: as usual in CLP, we will suppose to have a language $\Sigma_{\mathcal{C}}$ of atoms that are interpreted as constraints in a domain $\mathcal{C}$. A solver, $solv_{\mathcal{C}}$ represents the operational counterpart, and the type of inferences it is able to perform are typical of the chosen domain [7].

Events are expressed as

$$\mathbf{H}(\mathit{Event}[, \mathit{Time}]) \tag{5}$$

where *Event* is a term describing the occurred event and *Time* is the time at which the event occurred. We omit the time parameter in some of the examples, for the sake of simplicity.

Events are mapped into a **H** predicate. The history **HAP** grows monotonically as new events are recorded. A **H** atom is always ground: when an event happens, we suppose to be given all the significant information about it.

Expectations (positive and negative) are hypotheses about the (future) events; the events that will actually happen may then fulfill or not the expectations.

The syntax of expectations is the following:

$$\mathbf{E}(\mathit{Event}[, \mathit{Time}]) \tag{6}$$

$$\mathbf{EN}(\mathit{Event}[, \mathit{Time}]) \tag{7}$$

for, respectively, positive and negative expectations. **E** is a positive expectation about an event (the event is *expected* to happen) and **EN** is a negative expectation, (i.e., the event is *expected not* to happen in order to fulfill the protocols).

Note the difference between $\neg\mathbf{E}(X)$ and $\mathbf{EN}(X)$. The first expresses the fact that there is no expectation about the happening of event X (yet, if the event happens, no expectation will be disconfirmed), while the second expresses the fact that the event is expected not to happen.

Expectations can have non-ground terms as arguments. Variables in an **E** atom are always existentially quantified. We often need to share variables between expectations; thus the scope of the existentially quantified variables in abducibles is the whole set of expectations.

On the other hand, **EN** atoms represent something that hopefully will not (ever) happen. Variables in a **EN** atom are universally quantified if they are not shared with an **E** atom. To sum up

– variables in **E** atoms are always existentially quantified with scope the entire set of expectations
– the other variables, that occur only in **EN** atoms are universally quantified.

In the following, we will use the notation **EXP** for the set of expectations.

Finally, the variables in expectations can be constrained by means of CLP constraints; for example, the **EXP** may contain

$$\mathbf{E}(p(X), T), X > 10, T < 20$$

meaning that the event $p(X)$ is expected for some X greater than 10, in some time before the instant 20 (i.e., $\exists_{X,T}\mathbf{E}(p(X),T) \wedge X > 10 \wedge T < 20$). On universally quantified variables, *quantifier restrictions* [28] can be imposed. For example, the formula:

$$\mathbf{EN}(q(X),T), T > 20$$

means that the event $q(X)$ is expected not to happen for all values of X in all times before instant 20 (no expectation holds after time 20). This means that

$$\forall_X, \forall_{T>20}, \mathbf{EN}(q(X),T)$$

which means, according to Bürckert [28],

$$\forall_X, \forall_T, T > 20 \Rightarrow \mathbf{EN}(q(X),T).$$

We restrict ourselves to *unary* quantifier restrictions, meaning that they involve only *one* variable. Formally, we give the following definition.

Definition 8. *A* quantifier restriction *for a universally quantified variable X is a (unary) constraint $c(X)$ indicating the values that the variable X represents. If $QR(X)$ is the set of quantifier restrictions of X, then the following formula:*

$$\forall X_{QR(X)} F$$

holds iff the following holds:

$$\forall X, QR(X) \Rightarrow F.$$

Note the difference between quantifier restrictions and constraints. A constraint would mean that

$$\forall X, c(X) \wedge F$$

which is false if $c(X)$ is not satisfied for every possible X.

In the case of existentially quantified variables, however, the two coincide:

Definition 9. *A* quantifier restriction *for an existentially quantified variable X is a (unary) constraint $c(X)$ indicating the values that the variable X represents. If $QR(X)$ is the set of quantifier restrictions of X, then*

$$\exists X_{QR(X)} F \overset{def}{\Longleftrightarrow} \exists X, QR(X) \wedge F.$$

A.2 Syntax of the KB

We consider the KB to be a logic program. Its syntax is the following:

$$\begin{aligned} KB &::= [Clause]^{\star} \\ Clause &::= Atom \leftarrow Body \\ Body &::= ExtLiteral\ [\ \wedge ExtLiteral\ ]^{\star} \\ ExtLiteral &::= Literal \mid Expectation \mid Constraint \\ Expectation &::= [\neg]\mathbf{E}(Event\ [,T]) \mid [\neg]\mathbf{EN}(Event\ [,T]) \\ Literal &::= Atom \mid \neg Atom \mid true \end{aligned} \tag{8}$$

In a clause, the variables are quantified as follows:

– Universally, if they occur only in literals of kind **EN** (and possibly constraints), with scope the *Body*;
– Otherwise universally, with scope the entire *Clause*.

A.3 Syntax of $\mathcal{IC}_S$

The IC_S in $\mathcal{IC}_S$ are used to check that the combinations of generated expectations are consistent with the knowledge base and the history. Intuitively, IC_S are rules used to provide information about the *expected* outcomes.

The syntax of $\mathcal{IC}_S$ is as follows:

$$\begin{aligned}
\mathcal{IC}_S &::= [IC_S]^\star \\
IC_S &::= \chi \rightarrow \phi \\
\chi &::= (HEvent|Expectation)\ [\wedge BodyLiteral]^\star \\
BodyLiteral &::= HEvent|Expectation|Literal|Constraint \\
\phi &::= HeadDisjunct\ [\ \vee HeadDisjunct\ ]^\star|\bot \\
HeadDisjunct &::= Expectation\ [\ \wedge (Expectation|Constraint)]^\star \\
Expectation &::= [\neg]\mathbf{E}(Event\ [,T])\ |\ [\neg]\mathbf{EN}(Event\ [,T]) \\
HEvent &::= [\neg]\mathbf{H}(Event\ [,T]) \\
Literal &::= Atom\ |\ \neg Atom\ |\ true
\end{aligned} \tag{9}$$

Given a IC_S $\chi \rightarrow \phi$, χ is called the *body* (or the *condition*) and ϕ is called the *head* (or the *conclusion*).

Syntactic restrictions, scope and implicit quantification of variables
The rules of scope and quantification are as follows:

1. A variable must occur at least in an *Event* or in an *Expectation*.
2. The variables that occur in the body are quantified universally with scope the entire IC_S.
3. The variables that occur only in the head have as scope the disjunct they belong to and
   (a) if they occur in literals **E** or ¬**E** are quantified existentially;
   (b) otherwise they are quantified universally.
4. the quantifier ∀ has higher priority than ∃ when they have the same scope (e.g., literals will be quantified ∃∀ and not vice-versa).

There are several reasons why we decided to adopt this convention. Firstly, we wanted to keep the notation simple. We did not want to load the notation with explicit quantification symbols in the IC_S, and at the same time we wanted IC_S to have a simple reading, and an intuitive meaning associated. Secondly, we had to take into account compatibility issues: as described in the syntax of **EXP** (Section A.1), **E** atoms in **EXP** are existentially quantified, and **EN** atoms are universally quantified. The syntax of the KB is consistent with that of **EXP**, so atoms occurring in the *Head* of a IC_S are quantified in the same way (rule 3 above); for the same reason we impose the following *allowedness condition*:

Definition 10. *A IC_S is* Quantifier Allowed *if every variable occurring in the head in literals of type* **E**, ¬**E**, *or (in the body) in a negative, defined literal*

- *either does not occur in the body*
- *or it occurs in the body in a literal of type* **H**, **E**, ¬**E**.

Variables in Integrity Constraints, in ALP, are usually universally quantified with scope the whole Integrity Constraint, and rule 2 above states that the other variables in the IC_S are quantified accordingly, suggesting that IC_S are considered as particular Integrity Constraints.

Since we restrict ourselves to quantifier restrictions that are *unary* (see Section A.1), we impose an allowedness condition for IC_Ss:

Definition 11. *A IC_S is* Constraint Allowed *if*

- *all the variables that are universally quantified with scope the body do not occur in restrictions;*
- *for each restriction c occurring in the IC_S,*
  - *either c only involves variables that also occur in* **E**, ¬**E**, **H** *atoms*
  - *or it involves only* one *variable occurring in* **EN** *atoms and the others must occur in* **H** *atoms.*

A clause is Constraint Allowed *if the variables that are universally quantified with scope the body do not occur in restrictions, and all variables that occur in restrictions also occur in atoms* **E**.

An Abductive Logic Program is Constraint Allowed *if all the $\mathcal{IC}_S$ and all the clauses in the KB are constraint allowed.*

A.4 Inference rules for CLP unification

Concerning equality constraints, we suppose that:

- the constraint theory contains rules for the equality constraint
- the constraint solver contains the same rules for equality that are in the IFF proof procedure, i.e., the function *infer*(CS) performs the following substitutions in the constraint store:
  1. Replaces $f(t_1, \ldots, t_j) = f(s_1, \ldots, s_j)$ with $t_1 = s_1 \wedge \ldots \wedge t_j = s_j$.
  2. Replaces $f(t_1, \ldots, t_j) = g(s_1, \ldots, s_l)$ with $false$ whenever f and g are distinct or $j \neq l$.
  3. Replaces $t = t$ with $true$ for every term t.
  4. Replaces $X = t$ by $false$ whenever t is a term containing X.
  5. (a) Replaces $t = X$ with $X = t$ if X is a variable and t is not
     (b) Replaces $Y = X$ with $X = Y$ whenever X is a universally quantified variable and Y is not.
  6. (a) If $X = t \in CS_k$, applies the substitution X/t to the entire node.

Note that the symbol $=$ is overloaded since it is used for both representing the equality constraint in the constraint domain and the unification among terms. As it is usual in CLP [12], we distinguish between the two by means of types. Some predicates, the so called constraints, do not have a definition in the program, but their semantics is embedded in the constraint solver. Also, some of the functor symbols are not simply Herbrand terms, but are associated with symbols in the constraint domain (e.g., the symbol $+$ represents the addition, etc.), and variables can be associated to domains. In this way, the terms that are built from CLP functors and variables can be distinguished, and the correct $=$ can be applied.

Concerning $\neq$, we will again suppose that it is possible to syntactically distinguish the CLP-interpreted terms and atoms; the solver will perform some inference on the interpreted terms (typically, depending on the CLP sort), and will moreover contain the following rules, for uninterpreted terms:

1. Replaces $f(t_1, \ldots, t_j) \neq f(s_1, \ldots, s_j)$ with $t_1 \neq s_1 \vee \ldots \vee t_j \neq s_j$.
2. Replaces $f(t_1, \ldots, t_j) \neq g(s_1, \ldots, s_l)$ with $true$ whenever f and g are distinct or $j \neq l$.
3. Replaces $t \neq t$ with $false$ for every term t.
4. Replaces $X \neq t$ by $true$ whenever t is a term containing X.
5. (a) Replaces $t \neq X$ with $X \neq t$ if X is a variable and t is not
   (b) Replaces $Y \neq X$ with $X \neq Y$ whenever X is a universally quantified variable and Y is not.
6. (a) Replace $A \neq B$ with $false$ if A is a universally quantified variable without quantifier restrictions (i.e., $QR(A) = \emptyset$)
   (b) If A is a universally quantified variable with quantifier restrictions $QR(A) = \{c_1(A), \ldots, c_d(A)\}$, and B is not universally quantified, replace $A \neq B$ with $\neg c_1(B) \vee \ldots \vee \neg c_d(B)$.[5]
   (c) If A and B are universally quantified, with quantifier restrictions $QR(A)$ and $QR(B)$ then
      – if $\neg QR(A) \cap \neg QR(B) = \emptyset$, replace $A \neq B$ with $true$.[6]
      – otherwise, replace $A \neq B$ with false.

Note that we do not introduce explicitly a rule for existentially quantified variables. In this case, we delegate to the specific solver (we do not make assumptions on its behaviour). Some solvers can easily propagate constraints of this type. E.g., given $X \neq 1$ a Finite Domain solver can delete the value 1 from the domain of X. If the second term is not ground, the constraint is typically suspended (thus we do not have a transition). We will delay the $\neq$ constraint until it can be propagated by the given rules.

[5] Intuitively, A is universally quantified, thus it assumes every possible value except the ones forbidden by one of the c_i. Thus, the only way to satisfy this constraint is to impose that B assumes one of the values excluded for A.

[6] Intuitively, if the values taken by A have no intersection with the values taken by B, then $A \neq B$ is true.

The constraint solver also deals with quantifier restrictions. If a quantifier restriction (due to unification) gets all the variables existentially quantified, then we replace it with the corresponding constraint. E.g., if in the tuple we have two variables X and Y quantified as follows: $\exists Y, \forall_{X \neq 1}$, and variable X is unified with Y, we obtain that $\exists Y$, $Y \neq 1$ (the quantifier restriction $X \neq 1$ becomes a constraint on the variable Y).