Runtime Addition of Integrity Constraints in SCIFF*

Marco Alberti[1], Marco Gavanelli[2], and Evelina Lamma[2]

[1] CENTRIA, DI-FCT, Universidade Nova de Lisboa
Quinta da Torre - 2825-144 Caparica (Portugal)
m.alberti@fct.unl.pt
[2] ENDIF, Università di Ferrara
Via Saragat, 1 - 44100 Ferrara (Italy)
marco.gavanelli@unife.it
evelina.lamma@unife.it

Abstract. Abductive Logic Programming is a computationally founded representation of abductive reasoning. In most ALP frameworks, integrity constraints express domain-specific logical relationships that abductive answers are required to satisfy.
Integrity constraints are usually known *a priori*. However, in some applications (such as interactive abductive logic programming, multi-agent interactions, contracting) it makes sense to relax this assumption, in order to let the abductive reasoning start with incomplete knowledge of integrity constraints, and to continue without restarting when new integrity constraints become known.
In this paper, we propose a declarative semantics for abductive logic programming with addition of integrity constraints during the abductive reasoning process.
We propose an operational instantiation (with formal termination, soundness and completeness properties) and an implementation of such a framework based on the $\mathcal{S}$CIFF language and proof procedure.

1 Introduction

The philosopher Charles Sanders Peirce divides the reasoning schemes of humans into three types: *deduction* (reasoning from causes to effects), *induction* (synthesizing new rules from examples) and *abduction* (making hypotheses on possible causes from known effects).

Abductive Logic Programming [1] is a computational representation of abductive reasoning that allows to express relationships between effects and possible causes (by means of a logic program), as well as logical constraints over the hypotheses (integrity constraints). In ALP possible hypotheses are represented by special predicates (called *abducibles*) that are not defined, but can be hypothesized, as long as they satisfy the integrity constraints. A positive answer to a query posed to an ALP system will typically contain the set of abducibles

* A short version of this paper will appear as LIPIcs technical communication.

that are hypothesized in order for the query to succeed. Such an answer is called *abductive answer* in the ALP literature.

Several instances of ALP have been proposed in the literature [2–6], which differ for the logic language (and in particular for the type of abducibles and of integrity constraints that can be expressed).

While in many applications integrity constraints are known at the beginning of the reasoning process, it is sometimes useful to relax this assumption.

For instance, the classical application field of abductive reasoning is the diagnosis. However, in a realistic setting, a doctor does not simply listen to the patient enumerating all his/her symptoms. Instead, they have a bidirectional and multi-stage interaction: the doctor asks questions and refines his/her diagnosis based on the answers of the patient. So, there is the need to add information dynamically, often in the form of rules, that can rule out unrealistic sets of explanations.

In multi-agent reasoning, agents that employ abductive reasoning could exchange integrity constraints by a communication process, and continue operating with the newly acquired integrity constraints. In contracting, two agents try to reach an agreement and each agent tries to reach its goals. For example, one agent may want to buy a car, and the other wants to sell it; the first tries to get a price as low as possible, while the second has the opposite aim, and they negotiate on the model, the optionals, etc. Of course, each agent is unwilling to send all of its own knowledge, because the other would exploit it to get favourable conditions: if the buyer knew all the constraints of the seller, it would be able to compute the minimum possible price for the seller, and then propose such price. On the other hand, it is quite natural to tell some of the constraints only when needed, in order to speedup the negotiation, and avoid lingering on small variations of a meaningless solution. For instance, in case the buyer asks for a seat for children, the seller could reply: *"Ok, but you cannot install a children seat if you have the airbag"*, and the client has to take into consideration this constraint, when making new proposals. On the other hand, there is no reason for the seller to state such knowledge immediately from the beginning, as it still does not know if the buyer is interested at all in children seats.

An abductive reasoner might seek additional integrity constraints (possibly available from public repositories), depending on its current computation; for example, the number of integrity constraints could be very vast (as if one has to take into consideration all the EU rules for contracts), so only those strictly needed should be downloaded. Moreover, depending on the current state of the derivation one may choose to download regulations from one server or another: suppose I am deciding whether to buy a good from a service in Italy or in Portugal; I may first try to get the best price, but then check if the regulations of that country allow me to do such transaction. I will download the regulations of such country, check if my transaction is allowed, and, if it is not, I will backtrack and take the second choice.

Integrity constraints can also be obtained at runtime by means of an automated computational process; for instance, by inductive reasoning. Recently,

extensions of Inductive Logic Programming techniques (ILP for short), and the DPML algorithm in particular [7], have been proposed to learn integrity constraints from labelled traces (a database of events recording happened interactions or activities, or a database collecting events at run-time). The DPML target language is the $\mathcal{S}$CIFF abductive logic language [8], and this inductive approach has been experimented in various contexts (business processes, among others; see [9, 10]).

Such applications motivate an abductive logic programming framework where some of the integrity constraints are known in advance, and some are added to the abductive logic program during the computation.

In this paper we propose such an extension, and a declarative semantics for it.

We describe the instantiation and implementation of the extension in the $\mathcal{S}$CIFF abductive logic language [8]. $\mathcal{S}$CIFF is implemented using Constraint Handling Rules [11]; in particular, integrity constraints are mapped to CHR constraints. Thanks to the properties of CHR, adding a new constraint at runtime amounts to the single operation of *calling* the new constraint, i.e., it can be delegated to the CHR solver.

The paper is structured as follows. In section 2, we propose a declarative semantics of ALPs with dynamic addition of integrity constraints based on the $\mathcal{S}$CIFF language, and we show that it exhibits properties of termination, soundness and completeness. In section 3 we discuss some possible applications. Discussion of related work and conclusions follow.

2 Runtime addition of integrity constraints in $\mathcal{S}$CIFF

In this section, we give a semantics for the runtime addition of integrity constraints for the $\mathcal{S}$CIFF abductive logic language; however, the definitions can be easily generalized for other abductive logic languages.

2.1 $\mathcal{S}$CIFF language

In the following, we provide a brief introduction to the $\mathcal{S}$CIFF language, and, in particular, on how the knowledge base, integrity constraints and goals are expressed. A complete definition of the language is available in [8].

A $\mathcal{S}$CIFF program $\mathcal{P}$ is composed of

- a knowledge base $\mathcal{KB}$;
- a set $\mathcal{IC}_S$ of *static integrity constraints*.

Predicates In $\mathcal{S}$CIFF, predicates can be defined or abducibles, and can contain variables. Variables can be constrained as in Constraint Logic Programming [12].

Knowledge base In $\mathcal{S}$CIFF, the knowledge base is a set of clauses of the form

$$Head \leftarrow Body$$

where *Head* is an atom built on a defined predicate, and *body* is a conjunction of literals (built on defined predicates or abducibles) and CLP constraints.

Integrity constraints In $\mathcal{S}$CIFF, integrity constraints have the form

$$Body \rightarrow Head$$

where *Body* is a conjunction of abducible atoms, defined atoms and constraints, and *Head* is a disjunction of conjunctions of abducible atoms and CLP constraints, or *false*.

The allowed integrity constraints vary in ALP languages. For instance, they can state that a given conjunction of literals implies false (as in the Kakas-Mancarella [2] language); or that a conjunction of atoms implies a disjunction of atoms (as in the IFF language [3]).

Goal In $\mathcal{S}$CIFF, a *Goal* has the same syntax of the body of a clause in the knowledge base.

2.2 Declarative semantics

A declarative semantics for runtime addition of integrity constraints can be given as follows.

Given a program $\mathcal{P} = \langle \mathcal{KB}, \mathcal{IC}_S \rangle$ and a *goal* $\mathcal{G}$, a pair $\langle \Delta, \theta \rangle$, where Δ is a set of abducibles and θ is a substitution, is an *abductive explanation* for $\mathcal{G}$ with additional integrity constraints $\mathcal{IC}_D$ iff

1. $\mathcal{KB} \cup \Delta \models \mathcal{G}\theta$
2. $\mathcal{KB} \cup \Delta \models \mathcal{IC}_S \cup \mathcal{IC}_D$

where the symbol $\models$ is interpreted, in $\mathcal{S}$CIFF, as the 3-valued completion semantics [13]. If such conditions hold, we write $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\mathcal{IC}_D} \mathcal{G}$.

Example 1.

$$\begin{array}{r} p(X) \leftarrow q(X,Y), a(Y) \\ q(X,Y) \leftarrow r(Y), d(Y) \\ r(2) \end{array} \tag{1}$$

$$a(X) \rightarrow b(X) \vee c(X) \tag{2}$$

Given the knowledge base in equation (1) and the integrity constraint in equation (2), where $a/1$, $b/1$, $c/1$, and $d/1$ are abducibles, two abductive answers are possible for the query $p(1)$: $\{a(2), b(2), d(2)\}$ and $\{a(2), c(2), d(2)\}$.

However, with the additional integrity constraint

$$c(X), d(X) \rightarrow false,$$

only $\{a(2), b(2), d(2)\}$ is an abductive answer.

2.3 Operational semantics

The $\mathcal{S}$CIFF proof-procedure is a rewriting system that defines a proof tree, whose nodes represents states of the computation. A set of transitions that rewrite a node into one or more children nodes. $\mathcal{S}$CIFF inherits the transitions of the IFF proof-procedure [3], and extends it in various directions. We recall the basics of $\mathcal{S}$CIFF; a complete description is in [8], with proofs of soundness, completeness, and termination.

Each node of the proof is a tuple $T \equiv \langle R, CS, PSIC, \Delta \rangle$, where R is the resolvent, CS is the CLP constraint store, PSIC is a set of implications (called *Partially Solved Integrity Constraints*) derived from propagation of integrity constraints, and Δ is the current set of abduced literals. The main transitions, inherited from the IFF are:

Unfolding replaces a (non abducible) atom with its definitions;
Propagation if an abduced atom $a(X)$ occurs in the condition of an IC (e.g., $a(Y) \rightarrow p$), the atom is removed from the condition (generating $X = Y \rightarrow p$);
Case Analysis given an implication containing an equality in the condition (e.g., $X = Y \rightarrow p$), generates two children in logical or (in the example, either $X = Y$ and p, or $X \neq Y$);
Equality rewriting rewrites equalities as in the Clark's equality theory;
Logical simplifications other simplifications like (true $\rightarrow A$) $\Leftrightarrow A$, etc.

$\mathcal{S}$CIFF includes also the transitions of CLP [12, 14] for constraint solving.

We extend $\mathcal{S}$CIFF with an additional transition defined as follows, and we call the resulting proof procedure $\mathcal{S}\text{CIFF}_D$.

Add IC Given a node $T \equiv \langle R, CS, PSIC, \Delta \rangle$ and an integrity constraint ic, transition *addIC* generates one node $T' \equiv \langle R, CS, PSIC \cup \{ic\}, \Delta \rangle$.

This transition picks integrity constraints from a queue of dynamic integrity constraints. The transition is applicable to any node in the proof tree, and it can be executed whenever the queue is not empty. More integrity constraints can be added to the queue during the computation.

Successful derivation A successful $\mathcal{S}\text{CIFF}_D$ derivation for an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$, with additional integrity constraints $\mathcal{IC}_D$ and a goal $\mathcal{G}$ is a sequence of nodes where

- the root node is $\langle \mathcal{G}, \emptyset, \mathcal{IC}_S, \emptyset \rangle$
- each node is generated from the previous one by a $\mathcal{S}\text{CIFF}_D$ transition
- the leaf node is $N \equiv \langle \text{true}, CS, PSIC, \Delta \rangle$

From the leaf node, a substitution θ is derived, that

- replaces all variables in N that are not universally quantified by a ground term;
- satisfies all the constraints in the store CS and the implications in $PSIC$.

If such a derivation exists, we write $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash_{\mathcal{IC}_D}^{\langle \Delta, \theta \rangle} \mathcal{G}$.

2.4 Properties

$\mathcal{S}$CIFF$_D$ properties can be derived from $\mathcal{S}$CIFF properties, by showing that a $\mathcal{S}$CIFF$_D$ derivation (successful or not) for the program $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$ with a finite set of additional integrity constraints $\mathcal{IC}_D$ can be transformed in an equivalent one, where a node is the root node of a $\mathcal{S}$CIFF derivation for the ALP $\langle \mathcal{KB}, \mathcal{IC}_S \cup \mathcal{IC}_D \rangle$.

Due to how the *addIC* transition operates, the number of *addIC* transitions in a derivation is finite, if the set of additional integrity constraints is finite, because each *addIC* transition removes one integrity constraint from the queue, and the transition is not applicable when the queue is empty.

Proposition 1. *Let N_2 be the node generated from node N_1 by transition T_1, and N_3 be the node generated from node N_2 by* addIC*. Then, if N_4 is the node generated from node N_1 by* addIC*, transition T_1 is applicable to N_4, and the node N_5 generated from N_4 by T_1 is equal to N_3, modulo renaming of variables.*

$$N_1 \xrightarrow{T_1} N_2 \xrightarrow{\text{addIC}} N_3$$
$$N_1 \xrightarrow{\text{addIC}} N_4 \xrightarrow{T_1} N_5$$

Proof. Let $N_1 \equiv \langle R_1, CS_1, PSIC_1, \Delta_1 \rangle$.

From the statement of the proposition, we can write the other nodes as:

$N_2 \equiv \langle R_2, CS_2, PSIC_2, \Delta_2 \rangle$

$N_3 \equiv \langle R_2, CS_2, PSIC_2 \cup \{ic\}, \Delta_2 \rangle$

$N_4 \equiv \langle R_1, CS_1, PSIC_1 \cup \{ic\}, \Delta_1 \rangle$

Since N_4 only differs from N_1 for the presence of an additional IC, T_1 is applicable on N_4, and it produces the same elements in the resulting node N_5, with an additional integrity constraint. That is, modulo renamings of variables,

$N_5 \equiv \langle R_2, CS_2, PSIC_2 \cup \{ic\}, \Delta_2 \rangle$

Proposition 2. *Let D be a $\mathcal{S}CIFF_D$ derivation that has k applications of the* addIC *transition. Then there exists a derivation D' that has the following properties:*

- *the first k transitions of D' are the same applications of* addIC*;*
- *each node of D', starting the transitions from $k+1$ is equal to the corresponding node of D.*

Proof. Trivially, by proposition 1, D' is obtained from D by moving the *addIC* transition to the beginning.

Termination Being $\mathcal{S}$CIFF based on the 3-valued completion semantics, its termination is proven, as for SLDNF resolution [15], for acyclic knowledge bases and bounded goals and implications. Of course, programs may also terminate in other cases as well. Other abductive proof-procedures are based on other semantics and can address also non-stratified programs [16].

Intuitively, for SLD resolution a level mapping must be defined, such that the head of each clause has a higher level than the body. For $\mathcal{S}$CIFF, as well as for the IFF, since it contains integrity constraints that are propagated forward, the level mapping should also map atoms in the body of an integrity constraint to higher levels than the atoms in the head; moreover, this should also hold considering possible unfoldings of literals in the body of an integrity constraint [17]. An atom is bounded w.r.t. a level mapping if the level mapping is bounded over the ground instances of that atom; an implication is bounded if all its literals are. The following proposition is the $\mathcal{S}$CIFF termination result [8].

Proposition 3. *Let $\mathcal{G}$ be a query to an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$, where KB_S, $\mathcal{IC}_S$ and $\mathcal{G}$ are acyclic w.r.t. some level mapping, and $\mathcal{G}$ and all implications in $\mathcal{IC}_S$ are bounded w.r.t. the level-mapping. Then, every $\mathcal{S}$CIFF derivation for each instance of $\mathcal{G}$ is finite.*

Termination is not affected in $\mathcal{S}\text{CIFF}_D$, as long as the newly added integrity constraints do not violate the termination conditions.

Proposition 4. *Let $\mathcal{G}$ be a query to an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$, with additional integrity constraints $\mathcal{IC}_D$, where KB_S, $\mathcal{IC}_S \cup \mathcal{IC}_D$ and $\mathcal{G}$ are acyclic w.r.t. some level mapping, and $\mathcal{G}$ and all implications in $\mathcal{IC}_S \cup \mathcal{IC}_D$ are bounded w.r.t. the level-mapping. Then, every $\mathcal{S}CIFF_D$ derivation for each instance of $\mathcal{G}$ is finite.*

Proof. Consider a $\mathcal{S}\text{CIFF}_D$ derivation D, with root node $\langle \mathcal{G}, \emptyset, \mathcal{IC}_S, \emptyset \rangle$. Then consider a derivation D', equal to D, except that all the *addIC* transitions are applied at the beginning. The node resulting from the last *addIC* transition is $\langle \mathcal{G}, \emptyset, \mathcal{IC}_S \cup \mathcal{IC}_D, \emptyset \rangle$. With the given hypotheses, such node is the initial node of a $\mathcal{S}$CIFF derivation that is finite, by the $\mathcal{S}$CIFF termination result (proposition 3). Since, by proposition 2, D is equal to D' for the part that follows the last *addIC* transition, D is also finite.

Soundness Since $\mathcal{S}\text{CIFF}_D$ only differs from $\mathcal{S}$CIFF for the presence of *addIC* transitions, and *addIC* is only applicable for a non-empty set of additional integrity constraints, a $\mathcal{S}\text{CIFF}_D$ derivation for an empty set of additional integrity constraints is a $\mathcal{S}$CIFF derivation for the same ALP.

Therefore, the $\mathcal{S}$CIFF soundness result [8] can be expressed as follows.

Proposition 5. *Given an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$, if*
$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash_{\emptyset}^{\langle \Delta, \theta \rangle} \mathcal{G}$
then
$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models_{\emptyset}^{\Delta} \mathcal{G}\theta$

We now extend it to the case with a (non-empty) set of additional integrity constraints $\mathcal{IC}_D$:

Proposition 6. *Given an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$, if*
$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash_{\mathcal{IC}_D}^{\langle \Delta, \theta \rangle} \mathcal{G}$
then
$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models_{\mathcal{IC}_D}^{\Delta} \mathcal{G}\theta$

Proof. Let D be a successful $\mathcal{S}\text{CIFF}_D$ derivation. By proposition 2, for the part after the last *addIC* transition, D is equal to a derivation D' where all the *addIC* transitions are applied at the beginning. The node after the last *addIC* transition in D' is $\langle \mathcal{G}, \emptyset, \mathcal{IC}_S \cup \mathcal{IC}_D, \emptyset \rangle$. The rest of the D' transition is a $\mathcal{S}$CIFF derivation. By the $\mathcal{S}$CIFF soundness result (proposition 5),

$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\mathcal{IC}_D} \mathcal{G}\theta$

Completeness Due to the fact that, as explained earlier, a $\mathcal{S}\text{CIFF}_D$ derivation for an empty set of additional integrity constraint is also a $\mathcal{S}$CIFF derivation for the same ALP, the $\mathcal{S}$CIFF completeness result [8] can be expressed as follows.

Proposition 7. *Given an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$ and, for any ground set Δ such that $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\emptyset} \mathcal{G}$ there exists Δ' such that $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash^{\langle \Delta', \theta \rangle}_{\emptyset} \mathcal{G}$, and $\Delta'\theta \subseteq \Delta$*

The corresponding result for $\mathcal{S}\text{CIFF}_D$, with non-empty set $\mathcal{IC}_D$ is as follows.

Proposition 8. *Given an ALP $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$ and a set $\mathcal{IC}_D$ of integrity constraints, for any ground set Δ such that $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\mathcal{IC}_D} \mathcal{G}$ there exists Δ' such that $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash^{\langle \Delta', \theta \rangle}_{\mathcal{IC}_D} \mathcal{G}$ and $\Delta'\theta \subseteq \Delta$.*

Proof. By the $\mathcal{S}$CIFF completeness result (proposition 7), there exists a $\mathcal{S}$CIFF derivation whose root node is $N = \langle \mathcal{G}, \emptyset, \mathcal{IC}_S \cup \mathcal{IC}_D, \emptyset \rangle$, and whose abductive answer is $\langle \Delta, \theta \rangle$.

But N is a node in a successful $\mathcal{S}\text{CIFF}_D$ derivation, whose root node is $\langle \mathcal{G}, \emptyset, \mathcal{IC}_S, \emptyset \rangle$, and *addIC* is applied for all ICs in $\mathcal{IC}_D$.

2.5 Implementation

The $\mathcal{S}$CIFF abductive proof procedure was implemented in Prolog, using extensively the Constraint Handling Rules [11, 18] library. The implementation can be downloaded from the $\mathcal{S}$CIFF web site [19] and runs on SICStus and SWI Prolog.

Constraint Handling Rules (CHR) is a logic language devoted to define new constraint solvers; however, it has been used as a general language for many different applications, not all strictly related to constraints.

A new solver is defined in CHR by means of rules. There exist two main types of rules: propagation and simplification[3]. A propagation rule is of the form

$$label@ \qquad Head_1, \ldots, Head_n \Rightarrow Guard | Body$$

and means that, if the optional *Guard* and the *Head*s are true, then the *Body* must be true. Operationally, whenever a set of constraints are in the store, matching $Head_1, \ldots, Head_n$, the *Guard* is checked; if it evaluates to *true*, the *Body* is executed (as a Prolog goal). The *label* is optional and serves only as an identifier of the rule.

[3] There are also *simpagation* rules, that are not logically necessary, but are important for efficiency; we will not go into details for lack of space.

Simplification rules have a similar syntax:

$$label@ \qquad Head_1, \ldots, Head_n \Leftrightarrow Guard | Body$$

They state that if the *Guard* is true, then the conjunction $Head_1, \ldots, Head_n$ is equivalent to *Body*. Operationally, if $Head_1, \ldots, Head_n$ are in the store (and *Guard* is true), they are removed and substituted by *Body*.

$\mathcal{S}$CIFF represents most of its data structures as CHR constraints:

- an abducible atom $a(X)$ is represented with the CHR constraint `abd(a(X))`
- a (partially solved) integrity constraint $a(Y), q(Y) \rightarrow p(Y) \vee c(Y)$ is represented as the CHR constraint

$$\texttt{psic(}\underbrace{\texttt{[abd(a(Y)),q(Y)]}}_{Body}\texttt{,}\underbrace{\texttt{(p(Y) ; abd(c(Y)))}}_{Head}\texttt{)}$$

  The *Head* can be any Prolog goal (it has the same syntax).

The proof tree is explored in a depth-first fashion, using the Prolog stack for this purpose. Transitions are implemented as CHR rules; for example, transition *Propagation* is implemented with the following propagation CHR:

```
propagation @
    abd(A1),
    psic([abd(A2)|More],Head)
    ==> psic([A1=A2|More],Head).
```

Case Analysis handles the equality in the body of a PSIC

```
case_analysis @
    psic([A=B|More],Head)
    ==> impose A=B
        psic(More,Head)
    ;   % Open choice point
        impose A and B do not unify
```

and the logical simplification $(true \rightarrow A) \Leftrightarrow A$ manages implications with empty body:

```
logic_simplification @ psic([],Head) <=> call(Head).
```

Thanks to this implementation, adding a new integrity constraint is just a matter of *calling* the corresponding CHR constraint: if we want to dynamically add the integrity constraint (2) we execute the goal:

```
psic( [abd(a(X))], (abd(b(X));abd(c(X))) ).
```

In this way, the newly added integrity constraint is automatically subject to all the applicable transitions. Consider rule `propagation`: whenever two constraints matching the rule head (e.g., `abd(a(1))` and `psic([a(X)],b(X))`) are present in the CHR constraint store, the rule is fired, it generates `psic([a(X)=a(1)],b(X))`, that triggers `case analysis`, which in its turn generates two children nodes:

– one where unification is imposed between the abducible in the CHR constraint store and the abducible in the partially solved integrity constraint, and a new partially solved integrity constraint is imposed, with the abducible removed from the body.
– one where disunification between the abducible in the CHR constraint store and the abducible in the partially solved integrity constraint is imposed.

In the previous example, `psic([a(X)=a(1)],b(X))` is rewritten in the first case as $X = 1$ and $b(X)$ is executed; in the second case by imposing the CLP constraint $X \neq 1$.

The relevant point, here, is that rule `propagation` is fired whenever both the constraints (the abducible and the `psic`) are in the CHR store, regardless of which one entered the store first. So, if a partially solved integrity constraint is added by *addIC*, and some abducible in its body is already in the store, propagation will occur, as if the partially solved integrity constraint had been in the constraint store from the beginning of the computation.

3 Applications

In this section, we show some typical applications of abductive reasoning where runtime addition of integrity constraints can be of use.

3.1 A diagnosis scenario

Traditionally, diagnosis has been one of the most natural applications of abductive reasoning, as it amounts at making hypotheses about the possible causes of given symptoms. However, usually a diagnosis is not just a matter of listing all symptoms, and finding any explanation, but the doctor can ask questions, and refine his/her idea based on the received answers. In the same way, an expert system based on abductive reasoning should be able to receive more information dynamically, and continue its computation based on the new information available.

Consider a patient that shows up at the doctor's with stomach ache and spots on his face.

The abduction-based diagnosis system used by the doctor has the following relevant clauses in the knowledge base, where *flu*, *ulcera*, *some_drug*, and *some_food* are abducibles:

stomach_ache ← *flu*
stomach_ache ← *ulcera*
spots ← *some_drug*
spots ← *some_food*

The query posed by the doctor is *stomach_ache*, *spots*. The abductive answers, equally plausible at this stage, are {*flu*, *some_drug*}, {*flu*, *some_food*}, {*ulcera*, *some_drug*}, and {*ulcera*, *some_food*}. To further investigate, the doctor asks the patient if he ate *some_food*, or he is being treated with *some_drug*. The

patient says that he does not eat *some_food*, but he is taking some pills; however he knows that *some_drug* gives him *ulcera*. The doctor now has two pieces of information that are relevant for this patient only: she adds dynamically the integrity constraints

$some\_food \rightarrow false$

$some\_drug \rightarrow ulcera$

to the abductive logic program.

The abductive answers are now $\{flu, some\_drug, ulcera\}$, and $\{ulcera, some\_drug\}$. The doctor may prefer the latter, for minimality.

3.2 Contracting in Service-Oriented Architectures

In [20], we (with co-authors) proposed an automated contracting system for service-oriented architectures. In our architecture, a number of service providers expose their policies, expressed in the $\mathcal{S}$CIFF language: in particular, integrity constraints represent pieces of policies in form of rules (for instance, if a customer wants to pay by bank transfer, the amount is expected to be credited by one week after the order), and domain-specific knowledge is expressed as $\mathcal{S}$CIFF knowledge base clauses and, where appropriate, by ontologies (such as the concept of accepted customer). A user will also have policies expressed in the $\mathcal{S}$CIFF language. In this application, abducibles represent actions by the users or the service providers (such as a message containing the credit card number, or the delivery of digital content).

The heart of the system is the $\mathcal{S}$CIFF Reasoning Engine (SRE). It receives as input a service provider's and a user's policies, as well as a user goal (such as the expectation to receive an ebook by one week). The system queries the $\mathcal{S}$CIFF proof procedure with the goal, using the union of the policies as the abductive program. The abductive answer, if one exists, is a sequence of abducibles that represent service provider's and user's actions, that achieve the goal while complying with the service provider's and the user's policies.

Runtime addition of integrity constraints would allow starting the reasoning with a subset of the policies, and to incrementally add policies. The benefit is twofold: first, not to waste computational resources to search for solutions that are already ruled out by partial policies; second, the user or service providers may not want to disclose all of their policies unnecessarily, but only when potential contractors (based on partial policies) have been identified. In Protune [21], *provisional predicates* are ground atoms that can change their values as result of actions carried out by a peer (e.g., providing a credential). In our framework, it would be possible for agents to exchange (partial) policies expressed in the rich $\mathcal{S}$CIFF language.

3.3 Abductive reasoning on the Web

Many problems in constraint programming require a very large set of constraints, that cannot be propagated efficiently all at once. For example, integer linear programming relies on a linear solver, that solves very efficiently linear constraints

(in some cases, with polynomial algorithms), but cannot handle natively the integrality constraints (i.e., the fact that some of the variables are required to have integer values). Integrality can be dealt with by adding more linear constraints (for example, Gomory cuts), but the number of such constraints can be exponential. The usually adopted solution is to add those constraints during search: if in the current candidate solution some variable X is not integer, only those constraints that remove the non-integral value of X are imposed.

In the same way, the $\mathcal{S}$CIFF proof-procedure can deal with problems with a very large number of integrity constraints. For example, laws or regulations can be expressed in the $\mathcal{S}$CIFF language, and published on a web site, in some web-friendly format (like RuleML or the RIF). $\mathcal{S}$CIFF is currently able to download web content through the PiLLoW library [22] and interpret data published in RuleML, so it can download from web servers a set of rules, interpret them as integrity constraints, and produce abductive answers that satisfy them. On the other hand, the set of regulations could be very wide, and downloading all of them could be a waste of time and bandwidth. Thanks to the dynamic addition of integrity constraints, $\mathcal{S}$CIFF could download only those rules that involve the current set of abduced literals, impose them to rule out inconsistent explanations, and iteratively provide more and more refined solutions, in an interactive procedure.

3.4 Experiments

To show the effectiveness of the approach, we tested a simple benchmark problem, that is a simplified version of a contracting scenario of Section 3.2. One agent needs to interact with some web service, and choose one that is able to provide the expected reply. In this example, the agent will tell message m and will expect n as reply. The agent knows the address of a series of web services, given as facts:

$$known\_service(http://web.address.one/folder1/policy.ruleml).$$
$$known\_service(http://web.address.two/folder2/policy.ruleml).$$

In order to find the right service, the agent executes the following goal, where *tell* is abducible:

$$known\_service(Addr), download\_ic(Addr), tell(me, S, m), not(tell(S, me, A), A \neq n)$$

meaning that it will non-deterministically choose a service, download its integrity constraints, and then tell message m; it will fail if it gets any reply that is not n.

We generated 25^2 services, each with one integrity constraint

$$tell(Client, s, letter_1) \rightarrow tell(s, Client, letter_2)$$

where $letter_1$ and $letter_2$ are substituted with a ground term corresponding to one of the 25 letters of the alphabet.

We tried the goal on a slow network (mobile phone) and it took 173.350s to find the right service. As a comparison, a solution that first downloads the IC of all possible services before starting the solution takes 319.005s.

4 Related work

Among the many works on abduction in CHR by Christiansen and colleagues [23, 24], we emphasize an inspiring position paper [25], in which preliminary experiments are shown with integrity constraints mapped to CHR rules. In that work, Christiansen points out that through meta-rules it is possible to dynamically add integrity constraints. Here we extend the idea within the $\mathcal{S}$CIFF framework, which gives us a set of properties deemed crucial in the computational logic community. The operational semantics of $\mathcal{S}$CIFF is not based on that of CHR, but on the sound and complete semantics of the IFF [3]: this allowed us to prove those properties also for $\mathcal{S}$CIFF. In this paper, we extend these proofs for the dynamic addition of integrity constraints, reaching the objective pointed out by Christiansen, but with soundness and completeness results.

EVOLP [26] is a language to define logic programs able to evolve. A special atom $assert(Rule)$ can occur in the head or in the body of clauses; in case the stable model semantics assigns value *true* to some of these literals, the clause *Rule* is added to the program. Our instance can be considered as an evolving abductive program, in which only integrity constraints (and not clauses in the $\mathcal{KB}$) can be added, and based on the three-valued completion semantics, instead of the stable model semantics. Our language also features CLP constraints and, as the general CLP framework [12], it is parametric with respect to the specific sort. The proof procedure lets the user choose the associated solver, and two state-of-the-art solvers are available in the current implementation: CLP($\mathcal{R}$), on the real values, and CLP(FD), on finite domains. EVOLP is a component of the ACORDA prospective logic programming system [16], which also integrates abductive reasoning and preferences, to support interactive abductive logic programming, among other applications.

We can also easily extend the language in order to incorporate dynamic integrity constraints in the body of clauses, or in queries. Operationally, whenever an integrity constraint is part of the resolvent, the *addIC* transition would be applied. However, the impact of such extension on termination must be studied in future work. With reference to nested, dynamic ICs, and this extension of the SCIFF language, it is worth to mention that in the literature, a lot of work was devoted to the treatment of embedded implications (due to Miller, et al. see [27, 28] and McCarty, see [29]) based on the logic of Higher-Order Hereditary Harrop Formulas, a fragment of Intuitionistic logic. In this logic, and the λ system implemented [30], they allow arbitrary lambda terms with full higher-order unification, and extend the formula language with arbitrarily nested universal quantifiers and implications. In our case, we can add integrity constraints at runtime, rather than program clauses as they do. We can therefore support abductive reasoning in an extended set of constraints.

In CR-Prolog [31], new (consistency-restoring) rules can be added dynamically, as a part of Observe-Think-Act loop of an agent; if some inconsistency is detected then these constraints can be considered, according to their preferences. The semantics of CR-Prolog programs is defined as a transformation into abductive logic programs, where each consistency-restore rule has an abducible

associated with it, and holds (only) if such abducible is abduced. In our framework, dynamically added integrity constraints must be satisfied, independently of the abductive answer.

5 Conclusions

In this paper we proposed a declarative semantics for abductive logic programs where additional integrity constraints can be added at runtime. Such an extension can support interesting applications such as interactive abductive logic programming and contracting in service-oriented architecture.

We proposed an operational instantiation of such a framework as the $\mathcal{S}\text{CIFF}_D$ proof procedure, an extension of the $\mathcal{S}$CIFF abductive proof procedure, and we proved formal results of termination, soundness, and completeness for $\mathcal{S}\text{CIFF}_D$.

References

1. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation **2** (1993) 719–770
2. Kakas, A.C., Mancarella, P.: On the relation between Truth Maintenance and Abduction. In Fukumura, T., ed.: Proceedings of the 1st Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan, Ohmsha Ltd. (1990) 438–443
3. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33** (1997) 151–165
4. Denecker, M., De Schreye, D.: SLDNFA: An abductive procedure for abductive logic programs. Journal of Logic Programming **34** (1998) 111–167
5. Alferes, J.J., Pereira, L.M., Swift, T.: Well-founded abduction via tabled dual programs. In De Schreye, D., ed.: ICLP. (1999) 426–440
6. Wang, K.: Argumentation-based abduction in disjunctive logic programming. Journal of Logic Programming **45** (2000) 105–141
7. Lamma, E., Mello, P., Riguzzi, F., Storari, S.: Applying inductive logic programming to process mining. In Blockeel, H., Ramon, J., Shavlik, J.W., Tadepalli, P., eds.: ILP. Volume 4894 of Lecture Notes in Computer Science., Springer (2007) 132–146
8. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM Transactions on Computational Logics **9** (2008)
9. Chesani, F., Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Exploiting inductive logic programming techniques for declarative process mining. T. Petri Nets and Other Models of Concurrency **2** (2009) 278–295
10. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In Alonso, G., Dadam, P., Rosemann, M., eds.: BPM. Volume 4714 of Lecture Notes in Computer Science., Springer (2007) 344–359
11. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37** (1998) 95–138
12. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming **19-20** (1994) 503–582

13. Kunen, K.: Negation in logic programming. Journal of Logic Programming **4** (1987) 289–308
14. Jaffar, J., Maher, M., Marriott, K., Stuckey, P.: The semantics of constraint logic programs. Journal of Logic Programming (1994)
15. Apt, K.R., Bezem, M.: Acyclic programs. New Generation Computing **9** (1991) 335–364
16. Lopes, G., Pereira, L.M.: Prospective programming with ACORDA. In: Empirically Successful Computerized Reasoning (ESCoR'06) workshop at The 3rd International Joint Conference on Automated Reasoning (IJCAR'06), Seattle, USA (2006)
17. Xanthakos, I.: Semantic Integration of Information by Abduction. PhD thesis, Imperial College London (2003) Available at http://www.doc.ic.ac.uk/ ix98/PhD.zip.
18. Schrijvers, T., Demoen, B.: The K.U. Leuven CHR system: implementation and application. In Fruhwirth, T., Meister, M., eds.: First Workshop on Constraint Handling Rules. (2004)
19. Alberti, M., Chesani, F., Gavanelli, M.: The *SCIFF* abductive proof procedure (2004) http://lia.deis.unibo.it/research/sciff/.
20. Alberti, M., Cattafi, M., Gavanelli, M., Lamma, E., Chesani, F., Montali, M., Mello, P., Torroni, P.: Integrating abductive logic programming and description logics in a dynamic contracting architecture. In Hofmann, P., ed.: ICWS 2009: 2009 IEEE International Conference on Web Services, IEEE Computer Society, IEEE Computer Society Press (2009) 254–261
21. Coi, J.L.D., Olmedilla, D., Bonatti, P.A., Sauro, L.: Protune: A framework for semantic web policies. In Bizer, C., Joshi, A., eds.: International Semantic Web Conference (Posters & Demos). Volume 401 of CEUR Workshop Proceedings., CEUR-WS.org (2008)
22. Gras, D.C., Hermenegildo, M.V.: Distributed WWW programming using (Ciao-) Prolog and the PiLLoW library. TPLP **1** (2001) 251–282
23. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In Larsen, H.L., Kacprzyk, J., Zadrozny, S., Andreasen, T., Christiansen, H., eds.: FQAS, Heidelberg, Physica-Verlag (2000) 141–152
24. Christiansen, H., Dahl, V.: HYPROLOG: A new logic programming language with assumptions and abduction. In Gabbrielli, M., Gupta, G., eds.: ICLP. Volume 3668 of Lecture Notes in Computer Science., Springer (2005) 159–173
25. Christiansen, H.: Experiences and directions for abduction and induction using constraint handling rules. In: Workshop on abduction and induction AIAI'05, Edinburgh, Scotland (2005)
26. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In Flesca, S., Greco, S., Leone, N., Ianni, G., eds.: JELIA. Volume 2424 of Lecture Notes in Computer Science., Springer (2002) 50–61
27. Miller, D.: A logical analysis of modules in logic programming. Journal of Logic Programming **6** (1989) 79–108
28. Hodas, J.S., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Inf. Comput. **110** (1994) 327–365
29. McCarty, L.T.: Clausal intuitionistic logic I - fixed-point semantics. Journal of Logic Programming **5** (1988) 1–31
30. Nadathur, G., Miller, D.: An overview of lambda-prolog. In: ICLP/SLP. (1988) 810–827
31. Balduccini, M., Gelfond, M.: Logic programs with consistency-restoring rules. In: AAAI Spring 2003 Symposium. (2003) 9–18