Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 4–13
http://www.floc-conference.org/ICLP-home.html

RUNTIME ADDITION OF INTEGRITY CONSTRAINTS IN AN ABDUCTIVE PROOF PROCEDURE

MARCO ALBERTI [1] AND MARCO GAVANELLI [2] AND EVELINA LAMMA [2]

[1] CENTRIA, DI-FCT, Universidade Nova de Lisboa, Portugal

[2] ENDIF, Università di Ferrara, Italy

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, 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 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 [Kak93] is a computational representation of abductive reasoning that lets one 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 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 [Kak90, Fun97, Den98, Alf99, Wan00], 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

Key words and phrases: abduction, semantics, interactive computation, proof procedure.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.4

all his/her symptoms, but 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 [Lam07b], 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 [Alb08], and this inductive approach has been experimented in various contexts (business processes, among others; see [Che09, Lam07a]).

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 a declarative semantics for such an extension, and its implementation based on the $\mathcal{S}$CIFF abductive logic language [Alb08]. $\mathcal{S}$CIFF is implemented using Constraint Handling Rules [Frü98]; 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 describe the CHR-based implementation. In Section 4 we show some experimental results. 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

We first provide a brief introduction to the $\mathcal{S}$CIFF language. A complete definition is available in [Alb08].

$\mathcal{S}$CIFF is a Computational Logic language, whose predicates can be defined or abducibles, and can contain variables. Variables can be constrained as in Constraint Logic Programming [Jaf94a].

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

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

A $\mathcal{S}$CIFF 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.

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*.

$\mathcal{S}$CIFF computations are goal-directed. A $\mathcal{S}$CIFF *Goal* has the same syntax of the body of a clause in the knowledge base.

2.2. Declarative semantics

The declarative semantics for runtime addition of integrity constraints is given in terms of *abductive explanation* as follows.

Given a $\mathcal{S}$CIFF 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 in the 3-valued completion semantics [Kun87]. If such conditions hold, we write $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\mathcal{IC}_D} \mathcal{G}$.

Example 2.1.

$$\begin{array}{rcl} p(X) & \leftarrow & q(X,Y), a(Y) \\ q(X,Y) & \leftarrow & r(Y), d(Y) \\ r(2) & & \end{array} \tag{2.1}$$

$$a(X) \quad \rightarrow \quad b(X) \vee c(X) \tag{2.2}$$

Given the knowledge base in equation (2.1) and the integrity constraint in equation (2.2), where $a/1$, $b/1$, $c/1$, and $d/1$ are abducibles, two abductive explanations 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

$$c(X), d(X) \rightarrow \mathit{false},$$

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

2.3. Operational semantics

The $\mathcal{S}$CIFF proof-procedure consists of a set of transitions that rewrite a node into one or more child nodes. It encloses the transitions of the IFF proof-procedure [Fun97], and extends it in various directions. A complete description of $\mathcal{S}$CIFF proof procedure is in [Alb08], 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 also includes the transitions of CLP [Jaf94a, Jaf94b] for constraint solving.

To manage the run-time addition of integrity constraints, 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.

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 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

In this section, we state some relevant $\mathcal{S}$CIFF$_D$ properties. Due to lack of space, we omit the proofs, available in a companion technical report [Alb10].

Intuitively, $\mathcal{S}$CIFF$_D$ properties can be derived from $\mathcal{S}$CIFF properties, by showing that a $\mathcal{S}$CIFF$_D$ derivation for the program $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$ with a finite set of additional integrity constraints $\mathcal{IC}_D$ can be transformed into an equivalent one, where a node is the root node of a $\mathcal{S}$CIFF derivation for the ALP $\langle \mathcal{KB}, \mathcal{IC}_S \cup \mathcal{IC}_D \rangle$.

The following proofs are based on these formal properties:

Proposition 2.2. *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.*

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

Proposition 2.3. *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* addIC*;*
- *each node of D', starting the transitions from $k+1$ is equal to the corresponding node of D.*

2.4.1. *Termination.* Being $\mathcal{S}$CIFF based on the 3-valued completion semantics, its termination is proven, as for SLDNF resolution [Apt91], 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 [Lop06].

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 [Xan03].

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

Proposition 2.4. *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.*

2.4.2. *Soundness.* As usual, the soundness property states that the abductive answer computed in a successful derivation is correct according to the declarative semantics.

Proposition 2.5. *Given an ALP* $\langle \mathcal{KB}, \mathcal{IC}_S \rangle$*, if*
$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash^{\langle \Delta, \theta \rangle}_{\mathcal{IC}_D} \mathcal{G}$
then
$\langle \mathcal{KB}, \mathcal{IC}_S \rangle \models^{\Delta}_{\mathcal{IC}_D} \mathcal{G}\theta$

2.4.3. *Completeness.* The completeness result states that $\mathcal{S}$CIFF$_D$ can compute a subset of any ground abductive answer that is correct according to the declarative semantics.

Proposition 2.6. *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 exist* Δ' *and* θ *such that* $\langle \mathcal{KB}, \mathcal{IC}_S \rangle \vdash^{\langle \Delta', \theta \rangle}_{\mathcal{IC}_D} \mathcal{G}$ *and* $\Delta'\theta \subseteq \Delta$.

3. Implementation

The $\mathcal{S}$CIFF abductive proof procedure was implemented in Prolog, using extensively the Constraint Handling Rules [Frü98, Sch04] library. The implementation can be downloaded from the $\mathcal{S}$CIFF web site [SCI10] 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[1]. 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.

Simplification rules have a similar syntax:

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

and 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).

[1]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.

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 (2.2) 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 child 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.

4. Experiments

To show the effectiveness of the approach, we tested a simple benchmark problem, that is a simplified version of a contracting scenario. 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.

5. Related work

Among the many works on abduction in CHR by Christiansen and colleagues [Abd00, Chr05b], we emphasize an inspiring position paper [Chr05a], 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 [Fun97]: 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 [Alf02] 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 [Jaf94a], 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 [Lop06], 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 [Mil89, Hod94] and McCarty, see [McC88]) based on the logic of Higher-Order Hereditary Harrop Formulas, a fragment of Intuitionistic logic. In this logic, and the λ system implemented [Nad88], 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 [Bal03], new (consistency-restoring) rules can be added dynamically, as a part of an agent's Observe-Think-Act loop; 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.

6. Conclusions

In this paper we proposed a declarative semantics for abductive logic programs where additional integrity constraints can be added at runtime, based on the $\mathcal{S}$CIFF language.

We described $\mathcal{S}\text{CIFF}_D$, an extension of the $\mathcal{S}$CIFF proof procedure that supports runtime addition of integrity constraints, and we proved formal results of termination, soundness, and completeness for $\mathcal{S}\text{CIFF}_D$.

Such an extension can support interesting applications such as interactive abductive logic programming and contracting in service-oriented architecture.

References

[Abd00] Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In Henrik Legind Larsen, Janusz Kacprzyk, Slawomir Zadrozny, Troels Andreasen, and Henning Christiansen (eds.), *FQAS*, pp. 141–152. Physica-Verlag, Heidelberg, 2000.

[Alb08] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logics*, 9(4), 2008.

[Alb10] Marco Alberti, Marco Gavanelli, and Evelina Lamma. Runtime addition of integrity constraints in SCIFF. Tech. Rep. cs-2010-01, Università degli Studi di Ferrara, Dipartimento di Ingegneria, 2010. Available at http://www.unife.it/dipartimento/ingegneria/informazione/informatica/rapporti-tecnici-1.

[Alf99] José Júlio Alferes, Luís Moniz Pereira, and Terrance Swift. Well-founded abduction via tabled dual programs. In D. De Schreye (ed.), *ICLP*, pp. 426–440. 1999.

[Alf02] José Júlio Alferes, Antonio Brogi, João Alexandre Leite, and Luís Moniz Pereira. Evolving logic programs. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni (eds.), *JELIA*, *Lecture Notes in Computer Science*, vol. 2424, pp. 50–61. Springer, 2002.

[Apt91] Krzysztof R. Apt and Marc Bezem. Acyclic programs. *New Generation Computing*, 9(3/4):335–364, 1991.

[Bal03] Marcello Balduccini and Michael Gelfond. Logic programs with consistency-restoring rules. In *AAAI Spring 2003 Symposium*, pp. 9–18. 2003.

[Che09] Federico Chesani, Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Exploiting inductive logic programming techniques for declarative process mining. *T. Petri Nets and Other Models of Concurrency*, 2:278–295, 2009.

[Chr05a] Henning Christiansen. Experiences and directions for abduction and induction using constraint handling rules. In *Workshop on abduction and induction AIAI'05*. Edinburgh, Scotland, 2005.

[Chr05b] Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta (eds.), *ICLP*, *Lecture Notes in Computer Science*, vol. 3668, pp. 159–173. Springer, 2005.

[Den98] Marc Denecker and Danny De Schreye. SLDNFA: An abductive procedure for abductive logic programs. *J. Log. Program.*, 34(2):111–167, 1998.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[Fun97] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[Hod94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Inf. Comput.*, 110(2):327–365, 1994.

[Jaf94a] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[Jaf94b] Joxan Jaffar, Michael Maher, Kim Marriott, and Peter Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 1994.

[Kak90] A. C. Kakas and Paolo Mancarella. On the relation between Truth Maintenance and Abduction. In T. Fukumura (ed.), *Proceedings of the 1st Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan*, pp. 438–443. Ohmsha Ltd., 1990.

[Kak93] A. C. Kakas, R. A. Kowalski, and Francesca Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Kun87] Kenneth Kunen. Negation in logic programming. *J. Log. Program.*, 4(4):289–308, 1987.

[Lam07a] Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Inducing declarative logic-based models from labeled traces. In Gustavo Alonso, Peter Dadam, and Michael Rosemann (eds.), *BPM*, *Lecture Notes in Computer Science*, vol. 4714, pp. 344–359. Springer, 2007.

[Lam07b] Evelina Lamma, Paola Mello, Fabrizio Riguzzi, and Sergio Storari. Applying inductive logic programming to process mining. In Hendrik Blockeel, Jan Ramon, Jude W. Shavlik, and Prasad Tadepalli (eds.), *ILP*, *Lecture Notes in Computer Science*, vol. 4894, pp. 132–146. Springer, 2007.

[Lop06] Gonçalo Lopes and Luís Moniz Pereira. 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.

[McC88] L. Thorne McCarty. Clausal intuitionistic logic I - fixed-point semantics. *J. Log. Program.*, 5(1):1–31, 1988.

[Mil89] Dale Miller. A logical analysis of modules in logic programming. *J. Log. Program.*, 6(1&2):79–108, 1989.

[Nad88] Gopalan Nadathur and Dale Miller. An overview of lambda-prolog. In *ICLP/SLP*, pp. 810–827. 1988.

[Sch04] T. Schrijvers and B. Demoen. The K.U. Leuven CHR system: implementation and application. In T. Fruhwirth and M. Meister (eds.), *First Workshop on Constraint Handling Rules*. 2004.

[SCI10] The *SCIFF* abductive proof procedure, 2010. `http://lia.deis.unibo.it/research/sciff/`.

[Wan00] Kewen Wang. Argumentation-based abduction in disjunctive logic programming. *J. Log. Program.*, 45(1-3):105–141, 2000.

[Xan03] I. Xanthakos. *Semantic Integration of Information by Abduction*. Ph.D. thesis, Imperial College London, 2003. Available at `http://www.doc.ic.ac.uk/~ix98/PhD.zip`.