DLINE JOURNALS

Journal of Data Processing

Print ISSN: 2278 – 6481
Online ISSN: 2278-649X

JDP 2024; 14 (1)
https://doi.org/10.6025/jdp/2024/14/1/1-10

Creation of Abstract Syntax for Declarative Logic Programming

Marco alberti[1] and Marco Gavanelli and Evelina Lamma[2]
[1]CENTRIA, DI-FCT, universidade nova de lisboa, portugal
[2]ENDIF, universita di ferrara, italy

Received: 18 October 2023
Revised: 29 December 2023
Accepted: 8 January 2024
Copyright: with Author(s)

ABSTRACT

Abductive logic programming is a logical representation of abductive reasoning. Most ALP frameworks express domain-specific logical relationships that the abductive answers must satisfy. A priori, the integrity constraints are known. However, for some applications (e.g., Interactive Abduction Logic Programming, Multi-Agent Interactions, Contracting), it is reasonable to loosen this assumption so that the abductive reasoning process starts incompletely aware of the integrity constraints and continues without restarting when a new integrity constraint is known. In the present paper, we provide an abstract syntax for the declarative logic programming of abductive logic, with the addition of integrity constraint during the process of abductive reasoning, an operational implementation with formal termination, good and completeness properties, and an implementation based on SCIFF language and proof procedure.

Keywords: *Abductive Logic Programming, Abductive Reasoning, Declarative Logic Programming, Abstract Syntax*

1. Introduction

The philosopher Peirce divides the reasoning schemes of humans into three types: deduction (reasoning from causes to eects), 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 eects 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 dened, 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 dier 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 eld 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, but they have a bidirectional and multi-stage interaction: the doctor asks questions, and renes 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 rst 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 SCIFF 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 SCIFF abductive logic language [Alb08]. SCIFF is implemented using Constraint Handling Rules [Frnu98]; 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 **SCIFF** 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 SCIFF

*In this section, we give a semantics for the runtime addition of integrity constraints for the **SCIFF** abductive logic language; however, the definitions can be easily generalized for other abductive logic languages.*

2.1. SCIFF Language

*We first provide a brief introduction to the **SCIFF** language. A complete definition is available in* **[*Alb08*]**.

***SCIFF** 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 **SCIFF** program P is composed of*

- *a knowledge base KB;*
- *a set IC_S of static integrity constraints.*

*A **SCIFF** 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 dened predicates or abducibles) and **CLP** constraints.*

*In **SCIFF**, integrity constraints have the form: $Body \rightarrow Head$, where $Body$ is a conjunction of abducible atoms, dened atoms and constraints, and Head is a disjunction of conjunctions of abducible atoms and **CLP** constraints, or $false$.*

***SCIFF** computations are goal-directed. A **SCIFF** $Goal$ 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 **SCIFF** program $P = \langle KB, IC_S \rangle$ and a goal G, a pair $\langle \Delta, \theta \rangle$, where Δ is a set of abducibles and θ is a substitution, is an abductive explanation for G with additional integrity constraints IC_D iff*

(1) $KB \cup \Delta \models G\theta$

(2) $KB \cup \Delta \models IC_S \cup IC_D$

*where the symbol $\models$ is interpreted, in **SCIFF**, as in the 3-valued completion semantics* **[*Kun87*]**. *If such conditions hold, we write $\langle KB, IC_S \rangle \models^{\Delta}_{\mathcal{IC}_D} G$.*

Example 2.1.

$$p(X) \leftarrow q(X, Y), a(Y)$$

$$q(X, Y) \leftarrow r(Y), d(Y) \qquad (2.1)$$

$$r(2)$$

$$a(X) \rightarrow b(X) \vee c(X) \qquad (2.2)$$

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

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

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

2.3. Operational Semantics

The $SCIFF$ *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* $SCIFF$ *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 denitions;*

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 simplications: *other simplications like* $(true \rightarrow A) \Leftrightarrow A$, *etc.*

$SCIFF$ *also includes the transitions of* CLP [Jaf94a, Jaf94b] *for constraint solving.*

To manage the run-time addition of integrity constraints, we extend $SCIFF$ *with an additional transition defined as follows, and we call the resulting proof procedure* $SCIFF_D$.

Add-IC: *Given a node* $T \equiv \langle R, CS, PSIC, \Delta\rangle$ *and an integrity constraint* ic, *transition* $addIC$ *generates one node T0* $T' \equiv \langle R, CS, PSIC \cup \{ic\}, \Delta\rangle$.

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 $SCIFF_D$ *derivation for an* ALP $\langle KB, IC_S\rangle$, *with additional integrity constraints* IC_D *and a goal* G *is a sequence of nodes where*

- *The root node is* $\langle G, \theta, IC_{S,} \theta\rangle$
- *Each node is generated from the previous by a* $SCIFF_D$ *transition*
- *The leaf node is* $N \equiv \langle true, CS, PSIC, \Delta\rangle$

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

- *Replaces all variables in* N *that are not universally quantied by a ground term;*
- *Satises all the constraints in the store* CS *and the implications in* $PSIC$.

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

2.4. Properties

In this section, we state some relevant ***SCIFF***$_D$ *properties. Due to lack of space, we omit the proofs, available in a companion technical report* **[*Alb10*]**.

Intuitively, ***SCIFF***$_D$ *properties can be derived from* ***SCIFF*** *properties, by showing that a* ***SCIFF***$_D$ *derivation for the program* $\langle KB, IC_S\rangle$ *with a nite set of additional integrity constraints* $\boldsymbol{IC_D}$ *can be transformed into an equivalent one, where a node is the root node of a* ***SCIFF*** *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 SCIFF*$_D$ *derivation that has k applications of the addIC transition. Then there exists a derivation D' that has the following properties:*

- *The rst 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 SCIFF 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-stratied 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 SCIFF, 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 SCIFF$_D$, *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* $\langle KB, IC_S\rangle$, *with additional integrity constraints* IC_D, *where* KB, $IC_S \cup IC_D$ *and* $\mathcal{G}$ *are acyclic w.r.t. some level mapping, and* $\mathcal{G}$ *and all implications in* $IC_S \cup IC_D$ *are bounded w.r.t. the level-mapping. Then, every SCIFF*$_D$ *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 **SCIFF**$_D$ 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 **SCIFF** abductive proof procedure was implemented in Prolog, using extensively the Constraint Handling Rules **[Frnu98, Sch04]** library. The implementation can be downloaded from the **SCIFF** web site **[SCI10]** and runs on **SICStus** and **SWI** Prolog.*

*Constraint Handling Rules (**CHR**) is a logic language devoted to dene new constraint solvers; however, it has been used as a general language for many dierent 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 simplication[1]. A propagation rule is of the form*

$$label@ \quad Head_1,\dots,Head_n \Rightarrow Guard \mid Body$$

and means that, if the optional Guard and the Heads are true, then the Body must be true. Operationally, whenever a set of constraints are in the store, matching $Head_1,\dots,Head_n$, 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.

Simplication rules have a similar syntax:

$$label@ \quad Head_1,\dots,Head_n \Leftrightarrow Guard \mid Body$$

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

***SCIFF** 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.*

Journal of Data Processing Volume 14 Number 1 March 2024

The proof tree is explored in a depth-rst 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 simplication $(true \rightarrow A) \Leftrightarrow A$ *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 red, 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 rst case as* $X=1$ *and* $b(X)$ *is executed; in the second case by imposing the constraint* $X \neq 1$*.*

The relevant point, here, is that rule propagation is red whenever both the constraints (the abducible and the psic) are in the ***CHR*** *store, regardless of which one entered the store rst. 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 eectiveness of the approach, we tested a simple benchmark problem, that is a simplied 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 nd 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 nd 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 **SCIFF** framework, which gives us a set of properties deemed crucial in the computational logic community. The operational semantics of **SCIFF** 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 **SCIFF**. 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 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 specic 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** (**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 system implemented [**Nad88**], they allow arbitrary lambda terms with full higher-order unification, and extend the formula language with arbitrarily nested universal quantiers 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 satised, 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 **SCIFF** language.*

*We described **SCIFF**$_D$, an extension of the **SCIFF** proof procedure that supports runtime addition of integrity constraints, and we proved formal results of termination, soundness, and completeness for **SCIFF**$_D$.*

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

References

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

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

[3] Alberti, Marco, Gavanelli, Marco., Lamma, Evelina. (2010). Runtime addition of integrity constraints in SCIFF. Tech. Rep. cs-2010-01, Universitadegli Studi di Ferrara, Dipartimento di Ingegneria. Available at http://www.unife.it/dipartimento/ingegneria/informazione/informatica/rapportitecnici-1.

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

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

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

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

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

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

[10] Christiansen, Henning., Dahl, Verónica. (2005). 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.

[11] Denecker, Marc., De Schreye, Danny. (1998). SLDNFA: An abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2), 111-167.

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

[13] Fung, T. H.., Kowalski, R. A. (1997). *The IFF proof procedure for abductive logic programming*.

Journal of Logic Programming, 33(2), 151-165.

[14] Hodas, Joshua S.., Miller, Dale. (1994). Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2), 327-365.

[15] Jaar, J.., Maher, M. J. (1994). Constraint logic programming: *A survey. Journal of Logic Programming*, 19-20, 503-582.

[16] Jaar, Joxan, Maher, Michael, Marriott, Kim., Stuckey, Peter. (1994). The semantics of constraint logic programs. Journal of Logic Programming.

[17] Kakas, A. C.., Mancarella, Paolo. (1990). 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.

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

[19] Kunen, Kenneth. (1987). Negation in logic programming. *Journal of Logic Programming*, 4(4), 289-308.

[20] Lamma, Evelina, Mello, Paola, Montali, Marco, Riguzzi, Fabrizio., Storari, Sergio. (2007). 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.

[21] Lamma, Evelina, Mello, Paola, Riguzzi, Fabrizio., Storari, Sergio. (2007). 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.

[22] Lopes, Gonçalo., Pereira, Luís Moniz. (2006). 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.

[23] McCarty, L. Thorne. (1988). Clausal intuitionistic logic I - Fixed-point semantics. *Journal of Logic Programming*, 5(1), 1-31.

[24] Miller, Dale. (1989). A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2), 79-108.

[25] Nadathur, Gopalan., Miller, Dale. (1988). An overview of lambda-prolog. In ICLP/SLP, pp. 810-827.

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

[27] The SCIFF abductive proof procedure. (2010). Retrieved from http://lia.deis.unibo.it/research/sciff/.

[28] Wang, Kewen. (2000). Argumentation-based abduction in disjunctive logic programming. *Journal of Logic Programming*, 45(1-3), 105-141.

[29] Xanthakos, I. (2003). Semantic Integration of Information by Abduction (Ph.D. thesis). Imperial College London. Retrieved from http://www.doc.ic.ac.uk/~ix98/PhD