The CHR-based Implementation of the $\mathcal{S}$CIFF Abductive System

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

[1] CENTRIA - DI/FCT - Universidade Nova de Lisboa
Quinta da Torre - 2829-516 Caparica, Portugal
[2] ENDIF - Università di Ferrara
Via Saragat, 1 - 44100 Ferrara, Italy.
m.alberti@fct.unl.pt,{marco.gavanelli|evelina.lamma}@unife.it

Abstract. Abduction is a form of inference that supports hypothetical reasoning and has been applied to a number of domains, such as diagnosis, planning, protocol verification. Abductive Logic Programming (ALP) is the integration of abduction in logic programming. Usually, the operational semantics of an ALP language is defined as a proof procedure.
The first implementations of ALP proof-procedures were based on the meta-interpretation technique, which is flexible but limits the use of the built-in predicates of logic programming systems. Another, more recent, approach exploits theoretical results on the similarity between abducibles and constraints. With this approach, which bears the advantage of an easy integration with built-in predicates and constraints, Constraint Handling Rules has been the language of choice for the implementation of abductive proof procedures. The first CHR-based implementation mapped integrity constraints directly to CHR rules, which is an efficient solution, but prevents defined predicates from being in the body of integrity constraints and does not allow a sound treatment of negation by default.
In this paper, we describe the CHR-based implementation of the $\mathcal{S}$CIFF abductive proof-procedure, which follows a different approach. The $\mathcal{S}$CIFF implementation maps integrity constraints to CHR constraints, and the transitions of the proof-procedure to CHR rules, making it possible to treat default negation, while retaining the other advantages of CHR-based implementations of ALP proof-procedures.

1 Introduction

According to the philosopher Peirce [1], abductive reasoning is one of the basic inferences a reasoning agent (and a human in particular) uses. It is a type of hypothetical reasoning associated with finding explanations for a given evidence. Its most classical application is diagnosis: we are given a symptom of a patient, or a wrong behaviour of a machine, plus a set of rules explaining which illnesses might cause the symptom/misbehaviour, and we have to guess the right

cause. Besides diagnosis, abductive reasoning has been applied to a number of applications, like planning [2], protocol verification [3], etc.

Abductive Logic Programming (ALP) [4] is a language that embeds abductive reasoning into logic programming. In ALP, we have a set of predicates that have no definition, and are called *abducibles*. The truth of such predicates cannot be proven, but it can be assumed: the abductive derivation will provide in the computed answer the set of abduced hypotheses, together with the binding (the classical answer of Logic Programming languages). However, in typical applications, not all combinations of assumptions make sense: some illnesses are to be excluded beforehand, depending e.g. on the sex of the patient. For this reason, in ALP the user can typically define a set of rules, called *Integrity Constraints*, that must be satisfied by the set of hypotheses. The operational semantics of an ALP is typically defined as a proof-procedure. A number of proof-procedures have been proposed in the past for performing abductive reasoning; they are typically implemented as Prolog meta-interpreters [5–8].

A number of researchers have become interested in abductive reasoning because it deals in a simple and sound way with negation [9]. Literal $not(a)$ is rewritten as an integrity constraint $a \rightarrow \mathit{false}$, and then handled appropriately by the proof procedure. This type of negation is also called *negation by default*.

ALP has also been integrated with Constraint Logic Programming [6, 8, 10], in order to use both abductive reasoning and constraint propagation.

Kowalski et al. [11] studied the theoretical similarities between constraints and abducibles. Such similarity was later exploited for the implementation of abductive proof-procedures where abducibles are mapped to CLP constraints. For this purpose, a promising is Constraint Handling Rules (CHR) [12] a language designed to implement new constraints and constraint solvers in a simple and efficient way.

The first works on the implementation of abductive reasoning in CHR [13–16] implemented directly the integrity constraints into CHR rules: in a sense, CHR becomes also the language for writing integrity constraints. Thus, the user can write rules such as

$$\mathbf{p} \wedge \mathbf{q} \rightarrow r,$$

where $\mathbf{p}$ and $\mathbf{q}$ are abducible predicates and r can be either an abducible or a defined predicate. The interest of a CHR implementation is not only theoretical: thanks to the tight integration of CHR in the host language (which is often Prolog), those proof-procedures can seamlessly access built-in constructs and constraint solvers. This means that they have access to the innumerable libraries written in Prolog, and they can even *recurse* through external predicates: the abductive program can invoke Prolog predicates, and also meta-predicates (e.g., `findall`, `minimize`, ...), which can in their turn request the abduction of atoms, etc. A proof-procedure written in *CHR* benefits immediately from all the improvements of *CHR* engines, as recently happened with the Leuven CHR implementation [17]. Finally, those ALP which do not exploit abduction (or use abduction only in a limited subset of the application) do not suffer from the meta-interpretation overhead, but run at full speed.

However, a rule with a defined predicate in the antecedent is not allowed: these languages sacrifice negation by default on the altar of efficiency, which is a sensible thing to do in some applications, but it is not in others.

The $\mathcal{S}$CIFF proof-procedure [18] was developed in 2003 with an alternative CHR implementation, in which integrity constraints are first-class objects, and the proof-procedure can actively reason about them. In particular, we map abducibles into *CHR* constraints and implement the transitions of the operational semantics as *CHR* rules; in this way, the implementation follows very closely the operational semantics. Thanks to the sound operational semantics, $\mathcal{S}$CIFF has a sound treatment of default (and also explicit) negation. Thanks to the *CHR* implementation, $\mathcal{S}$CIFF is smoothly integrated with a constraint solver. From a language viewpoint, $\mathcal{S}$CIFF has unique features that do not appear in other abductive proof-procedures: it handles universally quantified variables both in the abducibles and in the integrity constraints; CLP constraints (treated as quantifier restrictions [19]) can be imposed both on existentially and on universally quantified variables.

$\mathcal{S}$CIFF has been continuously developed and improved in the past few years, and now it is smoothly integrated in graphical interfaces, semantic web applications; it is considerably faster, more robust, and provides more features.

In this paper, we show the implementation in CHR of the abductive proof-procedure $\mathcal{S}$CIFF, and we report about its recent improvements.

The rest of the paper is organised as follows. We first describe the $\mathcal{S}$CIFF abductive framework in Section 2. After some preliminaries on *CHR* (Section 3), we present the implementation of $\mathcal{S}$CIFF in *CHR* (Section 4). Discussion of related work (Section 5) and conclusions (Section 6) follow.

2 An abductive framework

Abductive Logic Programming is a family of programming languages that integrate abductive reasoning into logic programming. An ALP is a logic program, consisting of a set of clauses, that can contain in the body some distinguished predicates, belonging to a set $\mathcal{A}$ and called *abducibles*, (that will be shown in the following in **boldface**). The aim is finding a set of abducibles $\Delta \subseteq \mathcal{A}$ that, together with the knowledge base, is an explanation for a given known effect (also called *goal* $\mathcal{G}$):

$$KB \cup \Delta \models \mathcal{G}.$$

Also, Δ should satisfy a set of logic formulae, called *Integrity Constraints* IC:

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

E.g., if a patient has a headache, a physician may consult a knowledge base

$$\begin{array}{l} \mathit{headache} \leftarrow \mathbf{flu}. \\ \mathit{headache} \leftarrow \mathbf{migraine}. \\ \mathit{headache} \leftarrow \mathbf{meningitis}. \end{array}$$

and the abductive system will return one of the three explanations.

$\mathcal{S}$CIFF [18] is a language in the ALP class. CLP [20] constraints can be imposed on variables (which allows, for instance, to express that an event is expected to happen in a given time interval). For example, we might have an integrity constraint

$$\mathbf{flu} \rightarrow \mathbf{temp}(T), T < 39$$

saying that the explanation **flu** is acceptable only if the temperature of the patient is less than 39^oC. The computed answer includes in general three elements: a substitution for the variables in the goal (as usual in Prolog), the constraint store (as in CLP), and the set Δ of abduced literals.

$\mathcal{S}$CIFF was originally developed for the verification of interaction in multiagent systems [21, 22] and it is an extension of the IFF proof-procedure [7].

3 A brief introduction to Constraint Handling Rules

Constraint Handling Rules [12] (*CHR* for brevity hereafter) are essentially a committed-choice language consisting of guarded rules that rewrite constraints in a store into simpler ones until they are solved. *CHR* define both *simplification* (replacing constraints by simpler constraints while preserving logical equivalence) and *propagation* (adding new, logically redundant but computationally useful, constraints) over user-defined constraints.

The main intended use for *CHR* is to write constraint solvers, or to extend existing ones. However, the computational model of *CHR* presents features that make it a useful tool for the implementation of the $\mathcal{S}$CIFF proof-procedure.

There are three types of *CHR*s: *simplification*, *propagation* and *simpagation*.

Simplification CHRs. Simplification rules are of the form

$$H_1, \ldots, H_i \Longleftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \tag{1}$$

with $i > 0$, $j \geq 0$, $k \geq 0$ and where the multi-head $H_1, \ldots, H_i$ is a nonempty sequence of *CHR* constraints, the guard $G_1, \ldots, G_j$ is a sequence of built-in constraints, and the body $B_1, \ldots, B_k$ is a sequence of built-in and *CHR* constraints.

Declaratively, a simplification rule states that, if the guard is true, then the left-hand-side and the right-hand-side are equivalent.

Operationally, when constraint instances $H_1, \ldots, H_i$ in the head are in the store and the guard $G_1, \ldots, G_j$ is true, they are replaced by constraints $B_1, \ldots, B_k$ in the body.

Propagation CHRs. Propagation rules have the form

$$H_1, \ldots, H_i \Longrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \tag{2}$$

where the symbols have the same meaning of those in the simplification rules (1).

Declaratively, a propagation rule is an implication, provided that the guard is true. Operationally, when the constraints in the head are in the store, and the guard is true, the constraints in the body are added to the store.

Simpagation CHRs. Simpagation rules have the form

$$H_1, \ldots, H_l \backslash H_{l+1}, \ldots, H_i \Longleftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \tag{3}$$

where $l > 0$ and the other symbols have the same meaning and constraints of those of simplification *CHR*s (1).

Declaratively, the rule of Eq. (3) is equivalent to

$$H_1, \ldots, H_l, H_{l+1}, \ldots, H_i \Longleftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k, H_1, \ldots, H_l \tag{4}$$

Operationally, when the constraints in the head are in the store and the guard is true, $H_1, \ldots, H_l$ remain in the store, and $H_{l+1}, \ldots, H_i$ are replaced by $B_1, \ldots, B_k$.

For example, the constraint $\leq$ can be implemented in CHR by giving its base properties, namely the following rules:

$$A \leq A \Leftrightarrow true \tag{5}$$

$$A \leq B, B \leq A \Leftrightarrow A = B \tag{6}$$

$$A \leq B, B \leq C \Rightarrow A \leq C \tag{7}$$

where the symbol '=' stands for unification. The CHR engine rewrites the constraints in the store occurring as in the left-hand-side of the rules; for example, if the constraints $X \leq Y$, $Y \leq X$ are in the store, they are removed from the store and the variables X and Y are unified, as prescribed by rule 6. Note that on the left-hand-side of a CHR rule only constraints defined with CHR can appear: while the right-hand-side can contain any Prolog predicate (including CLP(FD) constraints, unifications, etc.), these elements cannot appear on the left-hand-side.

4 Implementation of the $\mathcal{S}$CIFF proof-procedure

One of the features obtained through a *CHR* implementation (avoiding meta-interpretation) is that the resolvent of the proof is directly represented as the Prolog resolvent. This allows us to exploit the Prolog stack for depth-first exploration of the tree of states. More importantly, this means that we extensively reuse the Prolog machinery, and that built-in predicates in the host Prolog system can be called from the user's Abductive Logic Programs. We remark again that this feature comes for free together with the CHR implementation, and is not easily available in metainterpreter-based implementations of abductive proof-procedures.

In the same way, the constraint store of the constrained abductive proof-procedure[3] is represented as the union of the CLP constraint stores. For the implementation of the proof-procedure, we used the CLP(FD) and CLP(B) libraries, available both on SICStus [23] and SWI Prolog [24] We also have a

[3] This constraint store, which contains CLP constraints over variables, should not be confused with the *CHR* constraint store, which is used for the implementation of the other data structures.

CHR-based solver on finite and infinite domains, and we defined an *ad-hoc* solver for reified unification. Recently, the interface between $\mathcal{S}$CIFF and the constraint solver has been re-engineered, and now it allows the developer to adopt any constraint solver that implements a given interface. In this way, the user can choose for each application which solver he/she wants to use; moreover, new solvers can be added with very limited effort. For example, the constraint solver on the reals, CLP($\mathcal{R}$) [25] has been integrated into $\mathcal{S}$CIFF: the new solver is based on the simplex algorithm (plus branch-and-bound), which is very efficient for linear constraints.

To the best of our knowledge, the other abductive proof-procedures implemented in CHR map abducibles to CHR constraints. Integrity constraints, instead, are often represented as CHR rules (typically, propagation rules) [13, 15]. Since a propagation CHR can have only CHR constraints in the multiple heads, the corresponding abductive proof-procedure can contain only abducibles in the precondition. This limitation forbids in the proof-procedure the implementation of default negation, that was one of the main motivations behind Abductive Logic Programming [9]. The operational semantics is then an extension of the operational semantics of CHR.

$\mathcal{S}$CIFF was developed following a different idea: we wanted increased flexibility in our language, while retaining the features that come for free with the CHR implementation. We first defined the declarative and operational semantics of $\mathcal{S}$CIFF as extensions of the IFF [7]. The operational semantics is given through a set of transitions that transform a state into another. The implementation, which maps integrity constraints, as well as the other relevant data strutures, to CHR constraints (rather than CHR rules) and transitions to CHR rules, follows the operational semantics very closely.

In the following, we first show some examples of transitions; the interested reader can find the complete list of transitions in a previous publication [18], together with the proofs of soundness and completeness of the $\mathcal{S}$CIFF proof-procedure. We then describe the implementation of some transitions in Section 4.2.

4.1 Examples of transitions

Given an abducible $\mathbf{a}(X)$ and an integrity constraint

$$\mathbf{a}(Y) \wedge p(Z) \wedge Y > Z \rightarrow r(Z)$$

transition *propagation* generates the following implication (that we call *Partially Solved Integrity Constraint* or PSIC for short):

$$X = Y \wedge p(Z) \wedge Y > Z \rightarrow r(Z) \tag{8}$$

Now, a transition *case analysis* generates two nodes of an OR-tree: in the first we consider the case $X = Y$, so the previous implication is transformed into

$$p(Z) \wedge X > Z \rightarrow r(Z),$$

in the second node, we consider the case that $X \neq Y$, and in this case the implication (8) is already satisfied.

Suppose we choose the first node, and that the knowledge base contains the definition of predicate $p(Z)$, e.g., as a fact $p(1)$. Transition *unfolding* generates the following implication:

$$X > 1 \rightarrow r(1)$$

Now, *case analysis* is again applied to the implication: in the first node we consider the case $X > 1$, while in the second $X \leq 1$. In the first case, the goal $r(1)$ is invoked.

These are just some examples of the transitions. $\mathcal{S}$CIFF contains transitions for handling correctly the various items (abducibles, expectations, happened events, CLP constraints, negation by default, explicit negation, etc.) in the $\mathcal{S}$CIFF language.

4.2 CHR implementation

The implementation of the transitions in CHR follows very closely the operational semantics. The various types of data are mapped to CHR constraints, while the transitions are mapped into CHR rules. For example, abducibles are represented as $abd(X)$; this means that abducibles can be directly used in the knowledge base, and CHR will take care of all the machinery necessary to abduce a new literal and propagate its consequences. For example, the clause

$$g(X) : -\mathbf{a}(X), b.$$

can be written as

```
g(X) :- abd(a(X)), b.
```

A (partially solved) integrity constraint

$$\mathbf{a}(X) \wedge p(Y) \rightarrow r(Z) \vee q(Z)$$

is mapped to the CHR constraint

```
psic([abd(a(X)),p(Y)],(r(Z);q(Z))).
```

As a first attempt, the propagation transition (together with case analysis) can be implemented via the CHR rule

```
abd(X), psic([abd(Y)|Rest],Head)
 ==>
copy(psic([abd(Y)|Rest],Head),psic([abd(Y1)|Rest1],Head1)),
 reif_unify(X,Y1,B),
 (B#=1, psic(Rest1,Head1) ; B#=0).
```

(9)

where `copy` performs a renaming of an atom (which also considers the various types of quantification in the $\mathcal{S}$CIFF [18], as well as CLP constraints attached

to the variables), `#=` is the finite domain equality constraint and `reif_unify` is a CHR implementation of *reified unification* [26].

`reif_unify` is a CHR constraint that declaratively imposes that either `B` $= 1$ and the first two arguments unify, or `B` $= 0$ and the two atoms do not unify; in logics, `reif_unify(X,Y,B)` is true iff

$$X = Y \leftrightarrow B = 1.$$

Note that some of the details are taken care of directly by CHR: if we have a set of abducibles and a set of PSICs we do not have to remember explicitly which PSICs have been tried with which abducibles (in order to avoid loops), as CHR itself does this work.

Note also that propagation is attempted only with the first element of the partially solved integrity constraint's antecedent, but this does not impact on what integrity constraints will be completely solved. For instance, given the integrity constraint $\mathbf{a}, \mathbf{b} \rightarrow \mathbf{c}$, if $\mathbf{b}$ and $\mathbf{a}$ are abduced in sequence, $\mathbf{b}$ will not be propagated as soon as it is abduced, but only after $\mathbf{a}$ has been abduced and propagated, and the partially solved integrity constraint $\mathbf{b} \rightarrow \mathbf{c}$ has been added to the constraint store; in the end, $\mathbf{c}$ will be abduced anyway. In this way, we ensure that each atom is propagated only once with each integrity constraint, without a need to keep track of previous propagations.

A number of improvements can be done to rule (9). First of all, CHR uses efficient indexing and hash tables to avoid checking all the possible pairs of CHR constraints. Sadly, rule (9) does not exploit such features of CHR. Note that the constraints in the antecedent of the propagation *CHR* do not share any variable, thus the CHR engine has to try each possible pair of constraints of types `abd` and `psic`, while, intuitively, one should try only those pairs whose arguments may unify. A first idea would be to rewrite the transition as:

```
abd(X), psic([abd(X)|Rest],Head)
        ==> ...
```

which would use CHR hash tables much more efficiently, but it would propagate only when the arguments are already ground or bound to the same term. This would be a very lazy propagation, that does not exploit the reified unification algorithm.

However, since abducibles are atoms, they always have a main functor, thus the argument of `abd` is always a term, which can contain variables, but it cannot be a variable itself. It is sensible to exploit the main functor for improving the selection of candidates. We represent each abducible as a CHR constraint with two arguments, where the first argument contains a ground term used to improve the hashing: in the current implementation, it is a list containing the main functor and its arity. The code for abducing an atom X is then:

```
abd(X) :- functor(X,F,A), abd([F,A],X).
```

Now, the *propagation transition* can be implemented with the *CHR* propagation rule:

```
abd(F,X), psic([abd(F,Y)|Rest],Head)
 ==>
 fn_ok(X,Y) |
 copy(psic([abd(Y)|Rest],Head),psic([abd(Y1)|Rest1],Head1)),
 reif_unify(X,Y1,B),
 (B#=1, psic(Rest1,Head1) ; B#=0).
```
(10)

i.e., only those pairs with identical first argument (i.e., abducibles that share the same functor and arity) are tried. `fn_ok` is a predicate that checks if the two arguments can possibly unify, and is also used for improving efficiency.

Many of the transitions of $\mathcal{S}$CIFF open a choice point, as we can see from the example of Eq. (10). However, in case `reif_unify` immediately yields 0 or 1, there is no point in opening a choice point. Otherwise, one could delay the disjunction, in order to open choice points as late as possible, hoping that other transitions might constrain the value of the `B` variable, possibly making it ground. In other words, it would be more desirable to delay as much as possible the non-deterministic transitions (those opening choice points), while expediting the deterministic ones (those that do not open choice points). One reason is that the deterministic may fail, and in this case the choice points opened by nondeterministic choices would be useless.

In order to implement the delay mechanism, we defined a CHR constraint '`nondeterministic`' that holds, as argument, a non-deterministic goal. In the previous example, the propagation transition is actually rewritten as

```
abd(F,X), psic([abd(F,Y)|Rest],Head)
==>
fn_ok(X,Y) |
copy(psic([abd(Y)|Rest],Head),psic([abd(RenY)|RenRest],RenHead)),
reif_unify(X,RenY,B),
(B == 1 -> psic(RenRest,RenHead) ;
 B == 0 -> true ;
 nondeterministic((B#=1, psic(RenRest,RenHead)) ; B#=0)).
```

i.e., we check if reified unification imposed a value on the boolean variable `B`, and we open a choice point only in case it did not. The choice point is not actually opened immediately, but it is declared in a CHR constraint.

Then, we defined a set of CHRs for dealing with `nondeterministic` constraints. We alternate a deterministic and a non-deterministic phase: initially, in the derivation, only deterministic transitions can be activated. Later, when the fixed point of the deterministic ones is reached, *one* non-deterministic transition can be applied, and we return to the deterministic phase. In CHR:

```
switch2det @ phase(nondeterministic), nondeterministic(G) <=>
    call(G),
```

```
    phase(deterministic).
switch2nondet @ phase(deterministic) <=> phase(nondeterministic).
```

where rule `switch2nondet` should be one of the last rules to be tried.

Transition Unfolding. Differently from HYPROLOG [15], integrity constraints can involve literals built on defined predicates. This allows for a sound treatment of default negation: a negative literal $not(a)$ is converted into an implication $a \rightarrow false$. Given a PSIC whose body contains a literal of a predicate defined in the KB, transition *unfolding* unfolds the literal:

```
psic([Atom|Rest],Head) <=>
is_defined_literal(Atom) |
findall(clause(Atom,Body),clause(Atom,Body),Clauses),
unfold(Clauses,psic([Atom|Rest],Head)).

unfold([],_).
unfold([clause(Atom,Body)|Clauses],psic([Atom1|Rest1],Head1)):-
  ccopy(psic([Atom1,Rest1],Head1),psic([Atom2|Rest2],Head2)),
  Atom = Atom2,
  append(Body,Rest2,NewBody),
  psic(NewBody,Head2),
  unfold(Clauses,psic([Atom1|Rest1],Head1)).
```

This might pose problems of termination: if the unfolded predicate is recursive, there exists an infinite branch in the derivation. For example, consider the IC:

$$\mathbf{a}(List), member(Term, List) \rightarrow \mathbf{b}(Term) \tag{11}$$

with the knowledge base:

$$member(X, [X|T]).$$
$$member(X, [Y|T]) : -member(X, T).$$
$$g : -\mathbf{a}([1, 2, 3]).$$

Intuitively, the goal g is true provided that we abduce $\mathbf{a}([1, 2, 3])$ and $\mathbf{b}(1)$, $\mathbf{b}(2)$, $\mathbf{b}(3)$. However, if we unfold predicate *member* in the IC (11) before the atom $\mathbf{a}([1, 2, 3])$ was abduced, the unfolding generates an infinite number of implications. For this reason, early versions of $\mathcal{S}$CIFF delay the unfolding after the other transitions, in the hope of binding some of the variables. In this particular example, if `member` is unfolded only after $\mathbf{a}([1, 2, 3])$ is abduced, the number of implications generated is equal to the number of elements in the list L, which is finite.

However, in other cases defined predicates provide just the value of a parameter, in this example, a deadline:

$$\mathbf{start}(a, T_a) \wedge deadline(D) \rightarrow \mathbf{end}(b, T_b) \wedge T_b \leq T_a + D$$

The knowledge base contains a simple fact *deadline*(5) stating that the deadline is 5 time units. In this case, if the unfolding of *deadline* is postponed after propagation of the $\mathbf{start}(a, T_a)$ event, it is repeated as many times as the number of **start** atoms that will be abduced, which might be a big number. For this reason, recent versions of $\mathcal{S}$CIFF unfold eagerly the predicates defined only by facts, and lazily the other predicates.

Results. The efficiency of $\mathcal{S}$CIFF has greatly improved with respect to earlier versions [27]. The following experiments were run on a 1.5GHz Pentium M 715 processor, 512MB RAM computer running SICStus 4.0.7.

Experiment	$\mathcal{S}$CIFF 2005	$\mathcal{S}$CIFF 2011
Auction Protocol	2.27s	0.37s
Block World	45.0s	15.7s
A^{I}LoWS Feeble conformance	84.4s	36.8s
A^{I}LoWS non-conformant	3.7s	3.3s

The aim of these experiments is not to compete with other abductive proof-procedures, but to show the improvements obtained taking into consideration the features of *CHR*. The version 2011 features improved hashing, eager unfolding, and other minor improvements. The experiments are real-life applications that we developed in $\mathcal{S}$CIFF: the proof of conformance of agents to an auction protocol, planning in the abductive event calculus, and A^{I}LoWS [28], a system based on $\mathcal{S}$CIFF for the conformance verification of web services to choreographies.

4.3 $\mathcal{S}$CIFF as a System

From a software engineering perspective, since its first prototypical implementation [27] $\mathcal{S}$CIFF has been greatly improved, and it is now a fully fledged development system (see Fig. 1). An integrated development environment for $\mathcal{S}$CIFF ALPs, implemented as an Eclipse plugin, is now available. Through a RuleML parser, ALPs can be obtained dynamically from the web. Animations of the output are possible through Scalable Vector Graphics (SVG), the W3C standard for vector graphics and animations. A Graphical User Interface displays relevant information to the user [3]. Facts can be added dynamically from a number of sources, including Linda blackboards, Apache log files, Jade Sniffer Agent output.

5 Related work

The $\mathcal{S}$CIFF abductive framework is derived from the IFF proof procedure [7], which it extends in several directions: dynamic update of the knowledge base by

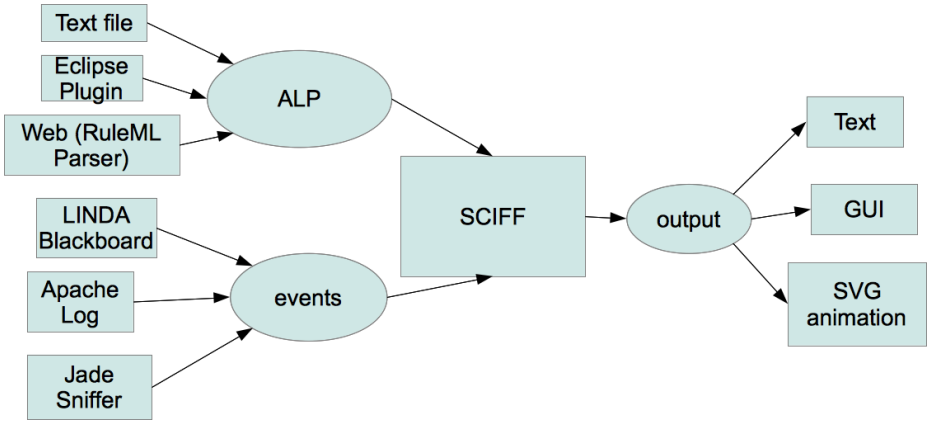

Fig. 1. Architecture of the $\mathcal{S}$CIFF system, illustrating some of the available inputs and outputs.

happening events, confirmation and disconfirmation of hypotheses, hypotheses with universally quantified variables, CLP constraints. Many other abductive proof-procedures have been proposed in the past; the interested reader can refer to the exhaustive survey by Kakas et al. [29].

Other proof-procedures deal with constraints; in particular we mention ACLP [6] and the $\mathcal{A}$-system [8], which are deeply focussed on efficiency issues.

Some conspicuous work has been done with the integration of the IFF proof-procedure with constraints [11]. Endriss et al. [10] present an implementation of an abductive proof-procedure that extends IFF [7] in two ways: by dealing with constraint predicates and with non-allowed abductive logic programs. The cited work, however, does not deal with confirmation and disconfirmation of hypotheses and universally quantified variables in abducibles, as ours does.

All of these proof-procedures are implemented as Prolog meta-interpreters. However, we believe that a *CHR* implementation has features that a meta-interpreted version cannot have, as we explained in the introduction.

Abdennadher and Christiansen [13] and Christiansen and Dahl [30] propose to exploit the *CHR* language to extend SICStus Prolog to support abduction more efficiently than with metainterpretation-based solutions. They represent abducibles as *CHR* constraints as we do, but they represent integrity constraints directly as *CHR* propagation rules, using the built-in *CHR* matching mechanism for propagation: this does not seem possible in our framework, in which we pose no limitations on the type of literals that occur in the conditions of integrity constraints. We also experimented with a similar implementation [14, 16], but it proved insufficient for our needs, as we needed a sound treatment of default negation and more flexibility in the quantification of variables.

6 Conclusions and future work

In this paper, we have presented the implementation of an abductive proof-procedure in *CHR*. We believe that the use of *CHR* in writing abductive proof-procedures has several advantages, compared to traditional approaches based on meta-interpretation. The first advantage is that $\mathcal{S}$CIFF benefits immediately from new implementations and improvements of *CHR* engines [31, 32, 17]. Another advantage is that the proof-procedure does not require meta-interpretation, which lets the user invoke built-in Prolog (meta)predicates within an Abductive Logic Program, without the need of contemplating explicitly their occurrence in the meta-interpreter. Also, Prolog is an instance of ALP (that does not use abduction): in $\mathcal{S}$CIFF, a Prolog program that does not use abduction runs at full speed, without the overhead of meta-interpretation.

An interesting extension of this work would be to integrate the two main ideas for implementing abduction in *CHR* in a unique framework. Each of the ideas have their own pros and cons: HYPROLOG, that implements integrity constraints as *CHR* rules, has less overhead, while $\mathcal{S}$CIFF, that maps integrity constraints into *CHR* constraints, is able to deal with default negation and is provably sound and complete. We are currently studying the idea of selecting syntactically the integrity constraints in an ALP in a preprocessing phase, and implementing each in the most efficient possible way, i.e., as *CHR* rules, whenever possible, or as *CHR* constraints when they contain defined predicates or CLP(FD) constraints.

Concerning confirmation, there are many possible extensions of this work, which we intend to pursue in the future. For instance, it would be worthwhile to let the user impose the failure of a branch of the reasoning tree, regardless of the confirmation or disconfirmation of the hypotheses made in the branch, in order to explore branches that the user finds more promising. We also intend to support a breadth-first exploration of the computation tree, as an alternative to the depth-first exploration of the current implementation. Besides, we believe that the formal framework would benefit from the introduction of a formalism to express priorities among the possible alternative hypotheses, in a given state of the computation.

Another direction of improvement could be towards better computational performance, possibly exploiting alternative efficient *CHR* implementations, such as the one proposed by Wolf [32].

Acknowledgments

This work has been supported by the European Commission within the e-Policy project (n. 288147).

References

1. Hartshorne, C., Weiss, P., eds.: Collected Papers of Charles Sanders Peirce, 1931–1958. Volume 2. Harvards University Press (1965)

2. Eshghi, K.: Abductive planning with the event calculus. In: Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, Cambridge, MA, MIT Press (1988)
3. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based tool. Applied Artificial Intelligence **20** (2006) 133–157
4. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation **2** (1993) 719–770
5. Denecker, M., Schreye, D.D.: SLDNFA: an abductive procedure for abductive logic programs. Journal of Logic Programming **34** (1998) 111–167
6. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive Constraint Logic Programming. Journal of Logic Programming **44** (2000) 129–177
7. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33** (1997) 151–165
8. Kakas, A.C., van Nuffelen, B., Denecker, M.: $\mathcal{A}$-System: Problem solving through abduction. In Nebel, B., ed.: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, Seattle, Washington, USA (IJCAI-01), Seattle, Washington, USA, Morgan Kaufmann Publishers (2001) 591–596
9. Eshghi, K., Kowalski, R.A.: Abduction compared with negation by failure. In Levi, G., Martelli, M., eds.: Proceedings of the 6th International Conference on Logic Programming, Cambridge, MA, MIT Press (1989) 234–255
10. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: The CIFF proof procedure for abductive logic programming with constraints. In Alferes, J.J., Leite, J.A., eds.: Proc. JELIA 2004. Volume 3229 of LNAI., Springer-Verlag (2004) 31–43
11. Kowalski, R., Toni, F., Wetzel, G.: Executing suspended logic programs. Fundamenta Informaticae **34** (1998) 203–224
12. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37** (1998) 95–138
13. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In Larsen, H., Kacprzyk, J., Zadrozny, S., Andreasen, T., Christiansen, H., eds.: FQAS, Flexible Query Answering Systems. LNCS, Warsaw, Poland, Springer-Verlag (2000) 141–152
14. Gavanelli, M., Lamma, E., Mello, P., Milano, M., Torroni, P.: Interpreting abduction in CLP. In Buccafurri, F., ed.: APPIA-GULP-PRODE Joint Conference on Declarative Programming, Reggio Calabria, Italy, Università Mediterranea di Reggio Calabria (2003) 25–35
15. Christiansen, H., Dahl, V.: HYPROLOG: A new logic programming language with assumptions and abduction. In Gabbrielli, M., Gupta, G., eds.: Proc. ICLP 2005. Volume 3668 of LNCS., Springer (2005) 159–173
16. Alberti, M., Chesani, F., Daolio, D., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interaction protocols in a logic-based system. Scalable Computing: Practice and Experience **8** (2007) 1–13
17. Schrijvers, T., Demoen, B.: The K.U. Leuven CHR system: implementation and application. In Frhwirth, T., Meister, M., eds.: Proc. CHR'04, Ulm, Germany (2004)
18. 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 Logic **9** (2008)
19. Bürckert, H.: A resolution principle for constrained logics. Artificial Intelligence **66** (1994) 235–271

20. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming **19-20** (1994) 503–582
21. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interactions using social integrity constraints. Electronic Notes in Theoretical Computer Science **85** (2003)
22. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An Abductive Interpretation for Open Agent Societies. In Cappelli, A., Turini, F., eds.: AI*IA 2003: Advances in Artificial Intelligence, Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence, Pisa. Volume 2829 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2003) 287–299
23. Carlsson, M., et al.: SICStus Prolog user's manual. Swedish Institute of Computer Science, Kista, Sweden. 4.0.7 edn. (2009) `www.sics.se/sicstus/`.
24. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming (2011) `http://arxiv.org/abs/1011.5332`.
25. Holzbaur, C.: OFAI clp(q,r) Manual. Austrian Research Institute for Artificial Intelligence, Vienna. 1.3.3 edn. (1995) TR-95-09.
26. Nuffelen, B.V.: Abductive Constraint Logic Programming: Implementation and Applications. PhD thesis, Katholieke Universiteit Leuven (2004)
27. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E.: The CHR-based Implementation of a System for Generation and Confirmation of Hypotheses. Number 2005-01 in Ulmer Informatik-Berichte (2005) 111–122
28. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M.: An abductive framework for a-priori verification of web services. In Maher, M., ed.: Proceedings of the Eighth Symposium on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy, New York, USA, Association for Computing Machinery (ACM), Special Interest Group on Programming Languages (SIGPLAN), ACM Press (2006) 39–50
29. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In Gabbay, D.M., Hogger, C.J., Robinson, J.A., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 5., Oxford University Press (1998) 235–324
30. Christiansen, H., Dahl, V.: Assumptions and abduction in Prolog. In Muñoz-Hernández, S., Gómez-Perez, J.M., Hofstedt, P., eds.: Workshop on Multiparadigm Constraint Programming Languages (MultiCPL'04), Saint-Malo, France (2004) Workshop notes.
31. Holzbaur, C., Frühwirth, T.: Compiling constraint handling rules into Prolog with attributed variables. In Nadathur, G., ed.: PPDP. (1999)
32. Wolf, A.: Adaptive constraint handling with CHR in Java. In Walsh, T., ed.: Principles and Practice of Constraint Programming - CP 2001. Volume 2239 of Lecture Notes in Computer Science., Paphos, Cyprus, Springer Verlag (2001) 256–270