An Experimental CLP Platform for Integrity Constraints and Abduction

Slim Abdennadher[1] and Henning Christiansen[2]

[1] Computer Science Department, University of Munich
Oettingenstr. 67, 80538 München, Germany
Slim.Abdennadher@informatik.uni-muenchen.de
[2] Department of Computer Science, Roskilde University,
P.O.Box 260, DK-4000 Roskilde, Denmark
henning@ruc.dk

Abstract Integrity constraint and abduction are important in query-answering systems for enhanced query processing and for expressing knowledge in databases. A straightforward characterization of the two is given in a subset of the language CHR$^\vee$, originally intended for writing constraint solvers to be applied for CLP languages. This subset has a strikingly simple computational model that can be executed using existing, Prolog-based technology. Together with earlier results, this confirms CHR$^\vee$ as a multiparadigm platform for experimenting with combinations of top-down and bottom-up evaluation, disjunctive databases and, as shown here, integrity constraint and abduction

1 Introduction

Constraint logic programming (CLP) [10] is established as an extension to logic programming that adds higher expressibility and in some cases more efficient query evaluation. CLP has also given rise to a field of constraint databases [14].

In the present paper, we suggest CLP techniques applied for defining database integrity constraints (ICs). ICs can be used for controlling (extensional or view) updates and provide additional flexibility for indirect definitions of database predicates by means of abductive definitions. As is well-known, abduction gives also rise to a kind of hypothetical queries that go beyond asking to the the logic consequence of the current database state.

A straightforward characterization of these features is provided in the CHR$^\vee$ language (to pronounce "CHR-or"), originally intended as a declarative language for efficient implementation of constraint solvers; CHR$^\vee$ extends the CHR language [8] with disjunctions in the body of its rules and is executable by existing CHR implementations.

In addition to provide a common framework, the results indicate that implementation methods that have been developed for efficient execution of CLP programs also can be applied for bottom-up evaluation in deductive databases with ICs and abduction. Such an approach is interesting for two reasons: There are cases where query answering requires the full versatility provided by a CLP environment and, secondly, it serves as an experimental platform for designing new advanced functionalities to improve systems for storage and management of large data quantities.

In a previous paper [3], CHR$^\vee$ has been demonstrated as a multiparadigm platform for combinations of top-down and bottom-up evaluation and disjunctive databases which, as shown here, also integrates with ICs and abduction.

The paper is organized as follows: The next section introduces a subset of the CHR$^\vee$ language, its declarative and operational semantics. Section 3 presents a translation of logic programs with integrity constraints into CHR$^\vee$ programs. In Section 4, we show how abductive frameworks can be formalized in CHR$^\vee$. Finally, we conclude with related work and a summary. No proofs are included for reasons of space.

2 CHR$^\vee$

The syntax and semantics of the subset of CHR$^\vee$ used in the present paper is briefly reviewed; see [3] for full details.

Explicit guards are omitted and no negation has been included in the language as it can be simulated by means of abduction and integrity constraints.

First-order terms, predicates, and atoms are defined in the usual way. We use two disjoint sorts of predicate symbols: One sort for *predefined* predicates and one sort for *free* predicates. Intuitively, predefined predicates are defined by means of a logical theory T_0 in such a way that statements of the form

$T_0 \models \forall(C_1 \wedge \cdots \wedge C_n \rightarrow \exists\bar{x}(C_{n+1} \wedge \cdots \wedge C_m))$,
where $\bar{x}$ is a subset of variables in $C_{n+1}, \ldots C_m$

are decidable. The legitimate forms of predefined predicates and their meaning is specified by a constraint domain for some CLP language and the sort of judgments shown above abstracts the behavior of an ideal constraint solver. Free predicates are those *defined* by a CHR$^\vee$ program.

The set of predefined predicates includes $=$ with the usual meaning of syntactic equality. An equation $c(t_1, \ldots, t_n)$=$d(s_1, \ldots, s_n)$ of two atoms stands for t_1=$s_1 \wedge \ldots \wedge t_n$=$s_n$ if c and d are the same symbols and for *false* otherwise. An equation $(p_1 \wedge \ldots \wedge p_n)$=$(q_1 \wedge \ldots \wedge q_m)$ stands for p_1=$q_1 \wedge \ldots \wedge p_n$=$q_n$ if $n = m$ and for *false* otherwise.

A *CHR$^\vee$ program* is a finite set of rules. There are two basic kinds of rules: *Simplification rules* $H \Leftrightarrow B$ and *propagation rules* $H \Rightarrow B$, where the *head* H is a non-empty conjunction of atoms and the *body* B a goal. A *goal* is a formula constructed from atoms by conjunctions and disjunctions in an arbitrary way; "*true*" denotes the empty conjunction and "*false*" the empty disjunction. A *query* is the same as a goal. Predefined predicates cannot occur in heads of rules.

A *simple goal* is one without disjunctions. G_{pre} denotes the conjunction of all predefined atoms of a simple goal G. Conjunctions can be permuted since conjunction is commutative and associative, and similarly for disjunctions.

The declarative semantics of CHR$^\vee$ is given by its embedding in first order logic. The logical meaning of a simplification rule $H \Leftrightarrow B$ is a logical equivalence:

$\forall \bar{x}\ (H \leftrightarrow \exists \bar{y}\ B)$. The logical meaning of a propagation rule $H \Rightarrow B$ is an implication: $\forall \bar{x}\ (H \rightarrow \exists \bar{y}\ B)$. In both cases, $\bar{y}$ refer to the variables of B that do not occur in H and $\bar{x}$ the remaining variables of the rule. The logical meaning of a CHR$^\vee$ program is the union of the logical meanings of its rules and T_0.

The operational semantics of CHR$^\vee$ is given by a transition system whose states are goals considered as a disjunction of *subgoals*. A subgoal G is *failed* if G_{pre} is unsatisfiable and a state is *failed* if all its subgoals are failed.

Given a CHR$^\vee$ program P, the transition relation $\mapsto_P$ is defined by the three kinds of derivations of Figure 1, **Simplify**, **Propagate**, and **Split**, applied to subgoals; the subscript P is left out when clear from context. A *derivation* for a query Q in a program P is a sequence $Q = S_0, S_1, \ldots$ of states with $S_i \mapsto S_{i+1}$, however so that no step can be applied to a failed subgoal and **Propagate** is not applied more than once to the same constraints in a given subgoal.[1] A *final state* in a derivation is either failed or a *successful* one to which no derivation step can be applied and which has at least one successful subgoal. Notice that the **Split** step ensures that successful subgoals always are simple.

Simplify

If $(H \Leftrightarrow B)$ is a variant with new variables of a rule of P with variables $\bar{x}$
and $T_0 \models \forall\ (G_{pre} \rightarrow \exists \bar{x}(H{=}H'))$
then $(H' \wedge G) \mapsto_P (H{=}H' \wedge B \wedge G)$

Propagate

If $(H \Rightarrow B)$ is a variant with new variables of a rule of P with variables $\bar{x}$
and $T_0 \models \forall\ (G_{pre} \rightarrow \exists \bar{x}(H{=}H'))$
then $(H' \wedge G) \mapsto_P (H{=}H' \wedge B \wedge H' \wedge G)$

Split

$$(G_1 \vee G_2) \wedge G \mapsto_P (G_1 \wedge G)\ \vee\ (G_2 \wedge G)$$

Figure1. Derivation Steps of CHR$^\vee$

To **Simplify** an atom H' means to replace it by an equivalent formula given by a rule in the program: A fresh variant $(H \Leftrightarrow B)$ of a simplification rule is made and H' replaced by the body B and the equation $H{=}H'$, provided $H{=}H'$ together with the other the predefined atoms in the current subgoal is satisfiable. A **Propagate** transition is like **Simplify**, except that it keeps the atoms H' in the state; its purpose is to add logically redundant information which may make it possible for other rules to be activated. Notice that simplification and propagation applies *matching* and not unification: H' must be an instance of H, i.e. the derivation step can only instantiate

[1] This to prevent trivial nontermination; see [2] for the formalization of a computation rule with the indicated properties.

variables of H but not variables of H'. (If unification were used instead of matching in a committed-choice language such as CHR$^\vee$, completeness would be lost.)

The **Split** transition can always be applied to a state containing a disjunction. No other condition needs to be satisfied. This transition leads to branching in the derivation in the sense that one subgoal is made into two, each of which needs to be processed separately. In Prolog, disjunction are processed by means of backtracking, one alternative is investigated and the second only if a failure occurs. For CHR$^\vee$, we need to investigate both branches in order to respect the declarative semantics; however, in an implementation this may be done, e.g., by backtracking (storing the results) or by producing copies of parts of the state to be processed in parallel or interleaved.

Spliting implies that a rule with a disjunction in its body is not just syntactic sugar for two clauses without disjunctions, i.e., $H \Leftrightarrow B_1 \vee B_2$ means something different than the combination of $H \Leftrightarrow B_1$ and $H \Leftrightarrow B_2$. In a derivation, the use of the rule with the disjunction means that both B_1 and B_2 occur in the subsequent state, whereas using the two other rules means a commitment to one of B_1 and B_2, i.e. one transition is chosen nondeterministically (in the sense of don't-care nondeterminism, i.e., without backtracking).

Example 1. Let P be the following CHR$^\vee$ program:

```
p(X) ⇔ q(X) ∨ r(X).
q(a) ⇔ false.
r(a) ⇔ true.
```

The evaluation of a query `p(a)` wrt. this program leads to the derivation

```
p(a) ↦ q(a) ∨ r(a) ↦ false ∨ r(a) ↦ false ∨ true
```

with a successful final state.

Simplification and propagation rules can be added for the predefined constraints in order to achieve more compact states, but from a semantic point this is unnecessary. In the examples that follow, we assume the equations in a subgoal be eliminated by the application of a suitable substitution and for brevity, we may sometimes leave out failed subgoals. The CHR$^\vee$ programs that we use can be executed directly by current implementations on CHR using Prolog's semicolon on right-hand sides and backtracking to produce all successful subgoals.

3 Logic programming with integrity constraints in CHR$^\vee$

As a first step towards a characterization of abduction, we show how a program P, written in a constraint logic language given by the predefined predicates of CHR$^\vee$, can be written as an equivalent CHR$^\vee$ program $\mathcal{C}P$ and how integrity constraints can be added to $\mathcal{C}P$. We distinguish predicates into *intensional*, defined by rules, and *extensional* ones, defined by finite sets of ground facts.

For each intensional predicate p/n defined by a number of clauses in P,

$$p(t_1^1, \ldots, t_n^1) \leftarrow b_1, \ldots, p(t_1^k, \ldots, t_n^k) \leftarrow b_k,$$

$\mathcal{C}P$ has a simplification rule called the *definition rule* for p/n of the form

$$p(x_1, \ldots, x_n) \Leftrightarrow (x_1 = t_1^1 \wedge \cdots \wedge x_n = t_n^1 \wedge b_1) \vee \cdots \vee (x_1 = t_1^k \wedge \cdots \wedge x_n = t_n^k \wedge b_k);$$

variables x_i do not occur in the original rules for p/n. For each extensional predicate p/n defined by a set of facts in P,

$$p(t_1^1, \ldots, t_n^1), \ldots, p(t_1^k, \ldots, t_n^k)$$

$\mathcal{C}P$ has a propagation rule, called the *closing rule for* p/n, of the form

$$p(x_1, \ldots, x_n) \Rightarrow (x_1 = t_1^1 \wedge \cdots \wedge x_n = t_n^1) \ \vee \ \cdots \ \vee \ (x_1 = t_1^k \wedge \cdots \wedge x_n = t_n^k)$$

In addition, $\mathcal{C}P$ has one propagation rule called *extensional introduction rule* of the following form, listing all facts of all extensional predicates of P.

$$\mathit{true} \Rightarrow f_1 \wedge \cdots \wedge f_n$$

Integrity constraints which can be added to $\mathcal{C}P$ are propagation rules of the form

$$e_1 \wedge \cdots \wedge e_n \Rightarrow b$$

where $e_1, \ldots, e_n$ are extensional atoms, b an arbitrary body.

The evaluation of a query will proceed in a top-down manner, unfolding it via the definition rules for the predicates applied, leading to a state giving sets of "hypotheses" about extensional predicates. The closing rules will prune this set so that only those hypothesis sets that are consistent with the facts of the original logic program are accepted. Notice that closing rules syntactically are special cases of integrity constraints and that they also serve as such, ensuring that no new extensional facts can be added.

Integrity constraints are not necessarily involved in the processing of a query but there are cases where the integrity constraint, as a kind of semantic optimization, can identify failures without consulting the actual extension (as embedded in closing and extensional introduction rules); this is shown in the example below. The extensional introduction rule ensures that no successful state can be reached without the integrity constraints being checked.

Example 2. The following CHR$^\vee$ program defines extensional `father`, `mother`, and `person` predicates and intensional `parent` and `sibling`. The integrity constraints state natural requirements that any set of extensional facts should satisfy. The predicate "$\neq$" is predefined representing syntactic nonequality.

```
% Definition rules
parent(P,C) ⇔ father(P,C) ∨ mother(P,C).
```

```
sibling(C1,C2) ⇔ C1≠C2 ∧ parent(P,C1) ∧ parent(P,C2).

% Extensional introduction rule
true ⇒
    father(john,mary) ∧ father(john,peter) ∧
    mother(jane,mary) ∧
    person(john,male) ∧ person(peter,male) ∧
    person(jane,female) ∧ person(mary,female) ∧
    person(paul,male).

% Closing rules
father(X,Y) ⇒ (X=john ∧ Y=mary) ∨ (X=john ∧ Y=peter).
mother(X,Y) ⇒ (X=jane ∧ Y=mary).
person(X,Y) ⇒
   (X=john ∧ Y=male) ∨ (X=peter ∧ Y=male) ∨
   (X=jane ∧ Y=female) ∨ (X=mary ∧ Y=female) ∨
   (X=paul ∧ Y=male).

% Integrity constraints
father(F1,C) ∧ father(F2,C) ⇒ F1=F2.
mother(M1,C) ∧ mother(M2,C) ⇒ M1=M2.
person(P,G1) ∧ person(P,G2) ⇒ G1=G2.
father(F,C) ⇒ person(F,male) ∧ person(C,S).
mother(M,C) ⇒ person(M,female) ∧ person(C,G).
```

The query `sibling(peter,mary)` will be unfolded to different subgoals involving `father` and `mother` hypotheses, some of which fail but `father(john,mary)` ∧ `father(john,peter)` survive and the query succeeds.

When a query is processed, the representation of extensional predicates by the extensional introduction rule introduces the facts into the state so that the integrity constraints are processed correctly, e.g., the query *true* succeeds, showing that the integrity constraints indeed are satisfied. The query `sibling(paul,mary)` will need the acceptance of `father(john,paul)` or `mother(jane,paul)` which are rejected by the closing rules and thus the query fails.

The query `father(X,Y)` ∧ `mother(X,Y)` can be brought to failure just by checking the integrity constraints.

```
father(X,Y) ∧ mother(X,Y) ↦
father(X,Y) ∧ mother(X,Y) ∧ person(X,male) ∧ person(Y,S) ↦
father(X,Y) ∧ mother(X,Y) ∧ person(X,male) ∧ person(Y,S) ∧
person(X,female) ∧ person(Y,S1) ↦
father(X,Y) ∧ mother(X,Y) ∧ person(X,male) ∧ person(Y,S) ∧
person(X,female) ∧ person(Y,S1) ∧ male=female ↦
false
```

The correctness properties of the transformation described above can be characterized as follows.

- For any positive Horn clause program P, the definition rules and extensional introduction rule of $\mathcal{C}P$ coincide with the Clark completion of P, and the closing rules are logically redundant. Thus the declarative semantics is preserved under the transformation.
- A successful subgoal contains a copy of the extensional part of P together with a satisfiable collection of predefined atoms that corresponds to a computed answer: equations characterize a substitution to the variables of the initial goal and if there are other atoms of predefined predicates, they serve as further constraints on those variable.
- If the database does not satisfy its integrity constraints, any initial goal has a failed derivation and with the formulation of a suitable computation rule we can get the result that any derivation will fail.
- Completeness is obvious for nonrecursive programs, whereas for recursive programs termination is not guaranteed. However, with a suitable computation rule, we can achieve termination analogous to a traditional CLP implementation for the language in which P is written. A completeness result can be formulated which collects the successful subgoals in a perhaps infinite derivation.

The formalism allows us also to define new predicates indirectly by means of integrity constraints that cannot be defined in a feasible way in positive constraint logic programs. It is sufficient to illustrate the principle by means of an example.

Example 3. We consider the task of extending the program of example 2 with an `orphan` predicate with the intended meaning that `orphan(X)` holds for any `X` which is a `person` but has no `father` or `mother`. A definition is required which is valid for any instance of the extensional predicates in the program, which means that an extensional listing of `orphan` facts is unacceptable. This can be expressed by adding the following three rules to the program.

`orphan(C)` $\Rightarrow$ `person(C,G).`
`orphan(C)` $\wedge$ `father(F,C)` $\Rightarrow$ $false$.
`orphan(C)` $\wedge$ `mother(M,C)` $\Rightarrow$ $false$.

The query `orphan(X)` results in a final state

`(X=john` $\wedge$ `orphan(john)` $\wedge$ E`)` $\vee$ `(X=jane` $\wedge$ `orphan(jane)` $\wedge$ E`)` $\vee$
`(X=paul` $\wedge$ `orphan(paul)` $\wedge$ E`)`

where E is the conjunction of the extensional facts in the program.

We see that the first rule defines a range for the `orphan`, giving rise to the possible instantiations of `X`, and the two next ones remove those values for `X` that have a `parent`.

This definition cannot be rewritten as a positive definition, and to express it in Prolog, we need to rely on the dubious procedural semantics of Prolog's approximation of negation-as-failure, e.g., as follows. `orphan(X):- person(X,_), \+father(X,_), \+mother(X,_).`

We return to this example in the next section.

4 Abduction and CHR$^\vee$

Abductive querying goes beyond what can be formulated by traditional queries concerning membership of the current state of a database: A goal, typically not implied by the database, is stated to the system and the answer describes ways how the goal could be achieved as a consequence of the database by additional hypotheses about the database state.

Abduction is usually defined as the process of reasoning to explanations for a given goal (or observation) according to a general theory that describes the problem domain of the application. The problem is represented by an abductive theory.

An abductive theory is a triple (P, A, IC), where P is a program, A is a set of predicate symbols, called abducibles, which are not defined (or are partially defined) in P, and IC is a set of first order closed formulae, called integrity constraints.

An abductive explanation or solution for a ground goal G is a set M of ground abducible formulae which when added to the program P imply the goal G and satisfy the integrity constraints IC, i.e.

$$P \cup M \models G \text{ and } P \cup M \models IC$$

In general, the initial goal as well as the solution set M may also contain existentially quantified abducibles together with some constraints on these variables [12,4,6], and such solutions are also produced by the method we describe below. We ignore the issue of minimality of solutions which often is required for abductive problem, e.g., diagnosis.

We consider here abduction for positive programs of the sort introduced in Section 3 and with integrity constraints formulated as CHR$^\vee$ rules as described. Abducibles must be chosen among the extensional predicates.

For such an abductive theory (P, A, IC), we define its translation into a CHR$^\vee$ program $\mathcal{C}(P, A, IC)$ similarly to translation of Section 3: The only difference is that the closing rule is left out for each abducible predicate; the extensional introduction rule contains as before all extensional facts, including possible initial facts for abducible predicates.

Example 4. The definition of the `orphan` predicate considered in example 3 is a special case of this translation with `orphan` considered as the only abducible predicate.

We consider a small example also used in [13].

Example 5. Consider an abductive framework with the following program and integrity constraint:

```
bird ← albatross.
bird ← penguin.
penguin ∧ flies → false.
```

Predicates `penguin, albatross` and `flies` are the only abducible. Obviously the abductive solutions for `bird` $\wedge$ `flies` is {`albatross, flies`}. These solutions are obtained by the following CHR$^\vee$ program.

```
bird ⇔ albatross ∨ penguin.
penguin ∧ flies ⇒ false.
```

With this program the evaluation of the goal `bird` $\wedge$ `flies` leads to the following derivation:

```
   bird ∧ flies
↦ (albatross ∨ penguin) ∧ flies
↦ (albatross ∧ flies) ∨ (penguin ∧ flies)
↦ (albatross ∧ flies) ∨ false
```

In the following, we show that our framework avoids problems with variables that exist in some abduction systems.

Example 6. We continue example 2 and let predicates `father` and `mother` (but not `person`) be abducible. The translation of this abductive framework is as shown in example 2 with the closing rules for `father` and `mother` removed.

The query `sibling(paul,mary)` succeeds in a final state containing two different abductive explanations (i.e., in different subgoals) `father(john,peter)` and `father(jane,peter)`. The abductive query `sibling(goofy mary)` fails since `person` is not abducible, and thus the closing rule for `person` will reject the hypothesis `person(goofy,-)`.

We can illustrate final states including variables by changing the example so that `person` becomes abducible, i.e., we remove the corresponding closing rule.

Now the query `sibling(goofy, mary)` leads to the following final state where E is the conjunction of extensional facts in the program:

```
(father(john,goofy) ∧ person(goofy,G) ∧ E)
∨
(mother(jane,goofy) ∧  person(goofy,G) ∧ E)
```

This first subgoal gives the abductive explanation for the `sibling` observation that `john` is the common `father` of `goofy` and `mary` and that the individual `goofy` must be a `person` whose precise gender is not necessary to specify. The second subgoal is similar, explaining `sibling` alternatively by means of a `mother` fact.

The query `sibling(goofy,mickey)` leads to a final state with two successful subgoals, one of which is

```
father(A,goofy) ∧ person(A,male) ∧
person(goofy,B) ∧ father(A,mickey) ∧
person(mickey,C) ∧ E)
```

The present approach to abduction avoids the problems with variables in abducibles that exist in some abduction algorithms. Three persons are necessary in the explanation, `mickey`, `goofy`, and their unknown `father`. The unknown `father` must be `male` and the gender of the others does not matter.

The correctness properties of the translation of abductive frameworks into CHR$^\vee$ program described above can be summarized as follows; we consider an abductive framework written as a CHR$^\vee$ program $\mathcal{C}(P, A, IC)$ and an initial goal G.

- For any successful subgoal $A \wedge C \wedge E$ in a derivation for G, where A are abducible atoms (not necessarily ground), C predefined, and E the extensional facts of P, and any grounding substitution σ which satisfies C, we have that $P \cup A\sigma \models G\sigma$ and $P \cup A\sigma \models IC$.
- If no successful subgoal exists in a derivation for G, there is no abductive explanation for (any instance of) G.
- Completeness is obvious for nonrecursive programs; for recursive programs the situation is similar to what we discussed for the translation of nonabductive programs into CHR$^\vee$.

Finally, we notice that this implementation in CHR$^\vee$ of abductive frameworks with integrity constraints is suited for implementing so-called explicit negation [16] so we can claim also to support negation in our framework. The technique is to introduce, for each predicate p/n, another abducible predicate *not-*p/n characterized by the integrity constraint $p(\bar{x}) \wedge$ *not-*$p(\bar{x}) \Rightarrow$ *false*.

5 Conclusion and related work

We have shown a straightforward characterization of important aspects of query-answering systems by means of CHR$^\vee$ programs. Programs of constraint logic languages with integrity constraints, important for describing "fine-grained" query evaluation and for constraint databases, can be written directly as CHR$^\vee$ programs that can be executed with current CHR technology. Abductive frameworks fit naturally into this model which provides an implementation that handles correctly a problem with variables in abducibles that exists in some earlier approaches to abduction, e.g., [11,7]. We can show that indirect characterization of predicates by means of integrity constraints and negation can be expressed also in straightforward ways. This shows that CHR$^\vee$ is useful as a specification language and an implemented, experimental framework for databases and query-answering mechanisms in general. Another important consequence of these results is to demonstrate that efficient implementation techniques for constraint logic programs, as embedded in the underlying CHR environment in deed is applicable for a variety of database applications and query answering mechanisms.

The approach is characterized by its simplicity; we do not need to use guards provided by CHR$^\vee$ and the procedural semantic can be described by a quite simple computational model. The constraint store is used to hold the extensional part of the database, including those new facts suggested in an abductive step, and integrity constraints serve as watchmen ready to strike as soon as an inconsistency is observed. The relation between integrity constraints and constraint logic has been suggested before in [5] in the shape of an incremental top-down evaluation method.

There are strong similarities between the work described in this paper and the work of Kowalski et al [13]. Both approaches originated from different starting points but the final result is very similar. Comparing with [13], their proof procedure may involve rewriting of complex formulae that likely can be optimized by methods similar to our use of an underlying CHR environment. Although not investigated in detail, we notice also a similarity between some of our examples and the semantic query optimizations described in [13,17].

CHR$^\vee$ can be seen as an extension of disjunctive logic programming with forward chaining [15] by constraints. The requirements for a constraint solver are essentially the same as those posed by traditional constraint logic programming systems with backward chaining.

It turns out that the Prolog-based implementations of CHR [9] are already able to evaluate CHR$^\vee$ programs, i.e. we use the disjunction of Prolog ";". Like in CHR, CHR$^\vee$ rules are compiled into Prolog clauses in textual order. More examples describing the use of CHR$^\vee$ are available in [1]. The examples can be executed online. To improve the efficiency of CHR$^\vee$, we plan to implement it at lower level such as modifying the underlying Prolog abstract machine. Furthermore, we plan to improve CHR$^\vee$ by adding strategies, i.e. declarative control on rules.

Acknowledgment: The second author's participation in this research is supported in part by the DART project funded by the Danish Research Councils, by the Danish Natural-Science Research Council, and the IT-University of Copenhagen.

References

1. Constraint handling rules online.
   `http://www.pms.informatik.uni-muenchen.de/~webchr/`
2. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer-Verlag, November 1997.
3. S. Abdennadher and H. Schütz. CHR$^\vee$: A flexible query language. *Flexible Query Answering Systems*, LNAI 1495, 1998.
4. H. Christiansen. Automated reasoning with a constraint-based metainterpreter. *Journal of Logic Programming*, 37(1-3):213–254, 1998. Special issue on Constraint Logic Programming.
5. H. Christiansen. Integrity constraints and constraint logic programming. In INAP Organnizing Committee, editor, *Proceedings of 12th International Conference on Applications of Prolog (INAP'99)*, pages 5–12, Tokyo, Japan, 1999.
6. H. Christiansen and D. Martinenghi. Symbolic constraints for meta-logic programming. *Journal of Applied Artificial Intelligence*, pages 345–368, 2000. Special Issue on Constraint Handling Rules.
7. H. Decker. An extension of sld by abduction and integrity maintenance for view updating in deductive databases. In *Proc. of JICSLP'96*, pages 157–169, 1996.
8. T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 1998.

9. C. Holzbaur and T. Frühwirth. A prolog constraint handling rules compiler and runtime system. *Journal of Applied Artificial Intelligence*, pages 369–388, 2000. Special Issue on Constraint Handling Rules.
10. J. Jaffar and M.J.Maher. Constraint logic programming: A survey. *Journal of logic programming*, 19,20:503–581, 1994.
11. A. C. Kakas and P. Mancarella. Database updates through abduction. In *Proc. 16th Int'l Conf. on Very Large Databases*, pages 650–661. Morgan Kaufmann, California, 1990.
12. A. C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 399–416, Cambridge, June 13–18 1995. MIT Press.
13. R. Kowalski, F. Toni, and G. Wetzel. Executing suspended logic programs. *Fundamenta Informaticae*, 34(3):203–224, 1998.
14. G.M. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer Verlag, 2000.
15. J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. MIT Press, 1992.
16. L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 102–106, Vienna, Austria, August 1992. John Wiley & Sons.
17. G. Wetzel and F. Toni. Semantic query optimization through abduction and constraint handling. *Flexible Query Answering Systems*, LNAI 1495:366–381, 1998.