Executable specifications for hypothesis-based reasoning with Prolog and Constraint Handling Rules

Henning Christiansen

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark

Abstract

Constraint Handling Rules (CHR) is an extension to Prolog which opens up a spectrum of hypothesis-based reasoning in logic programs without additional interpretation overhead. Abduction with integrity constraints is one example of hypothesis-based reasoning which can be implemented directly in Prolog and CHR with a straightforward use of available and efficiently implemented facilities.

The present paper clarifies the semantic foundations for this way of doing abduction in CHR and Prolog as well as other examples of hypothesis-based reasoning that is possible, including assumptive logic programming, hypotheses with priority and scope, and nonmonotonic reasoning.

Examples are presented as executable code so the paper may also serve the additional purpose of a practical guide for developing such programs, and it is demonstrated that the approach provides a seamless integration with existing constraint solvers.

Key words: Abduction, Abduction as deduction, Hypothesis-based reasoning, Logic programming

1 Introduction

Abduction in logic programming has proved to be a powerful technique for solving a range of complex problems, and a large number of approaches have been reported, see [1–4] for overview. Most known systems are based on meta-level architectures which means that abductive logic programs are interpreted by other logic programs rather than being compiled in an efficient way.

Preliminary version; final version to appear in *Journal of Applied Logic*, 2008 or -9

15 October 2008

We see abduction as one example within a class of more general hypothesis-based reasoning paradigms. The advantages of abduction are its straightforward declarative specification and the fact that a large class of relevant problems fits naturally into the framework: given an observation O, the task is to find an abductive explanation A which is a set of hypotheses which, when added to the current knowledge base, can explain O and does not introduce inconsistencies. The observation O may be observed symptoms of a malfunctioning system, and A a diagnosis for what may have caused O; or O can represent a desired goal to be achieved and A a set of requirements that must be present in order to reach that goal, e.g., a plan of primitive actions to be performed. While abduction concerns the creation of single sets of hypotheses that can explain the top query, there are other and more dynamic ways to apply hypotheses during a computation, as will be shown below.

In the present paper, we consider general hypothesis-based reasoning realized using available and efficiently implemented logic programming technology in a direct way, which eliminates any interpretative overhead. More specifically, we suggest PROLOG as an overall driver engine (and, as knowledge specification language) with Constraint Handling Rules, CHR, complementing the ability to manage hypotheses by declarative specifications and efficient implementation.

Example 1.1 The following program, which specifies and solves an abductive problem, is written in the syntax for CHR in SICStus and SWI PROLOG. Constraint predicates are distinguished from normal PROLOG predicates by declarations. Constraints of CHR play the role of abducibles and integrity constraints are written as CHR rules, e.g., as indicated below with '`==>`'.

```
:- chr_constraint professor/1, rich/1, has_good_students/1.
professor(X),rich(X) ==> fail.
happy(X):- rich(X).
happy(X):- professor(X), has_good_students(X).
```

Execution of the query `?- professor(peter), happy(peter)` results in a final constraint store (= abducible explanation) consisting of {`professor(peter)`, `has_good_students(peter)`}. The CHR rule excludes an explanation which includes `rich(peter)`, and thereby forcing the PROLOG engine to try the second clause in its eagerness to verify the query. □

The approach provides a seamless and efficient integration with all facilities of the underlying PROLOG and CHR system, including existing constraint solvers, in a way that makes it possible to go beyond strictly abductive reasoning. Futhermore, subtle problems with variables in abducible hypotheses in many earlier approaches do not arise here. The main weakness of using CHR and PROLOG for abduction in this way, when comparing with some other systems, is the limited use of negation.

By hypothesis-based reasoning, we refer to a space of problem solving and programming techniques in which logic programs are extended with a global state of hypotheses which interacts with the logic program and can be manipulated implicitly as in abductive reasoning or in more explicit ways, and which may or may not relate in all details to declarative specifications. It will be shown how abduction and other instances of hypothesis-based reasoning can be realized in straightforward ways with CHR and PROLOG, and we intend in this way also to indicate that there is a rich scope for developers to produce their own variants for different purposes.

Constraint Handling Rules were introduced in the early 1990es by Thom Frühwirth (primary reference [5] from 1998 provides background and early history), and is now available as extensions to major PROLOG systems, including SICSTUS and SWI. The idea of using CHR for abduction was originally suggested by Slim Abdennadher and the present author in 2000 [6]. The combination of PROLOG and CHR for abductive and other kinds of hypothesis-based reasoning has been developed together with Verónica Dahl since 2002 as a central collaborator, e.g., [7–9].

The present paper aims at giving a coherent presentation of the approach, clarifying the semantic foundations (that were left implicit in earlier publications) as well as exposing hypothesis-based reasoning with PROLOG and CHR as a powerful and flexible programming paradigm.

Examples have been checked using SICSTUS PROLOG [10]; we use in most cases the CHR syntax and facilities provided by SICSTUS PROLOG version 4 which is intended to be identical to that of SWI PROLOG [11]. Some extensions to CHR that we describe below are implemented using specifics of SICSTUS PROLOG version 3, which differs in some details and has a larger collection of low-level facilities, so transfer to version 4 may not be trivial in all cases.

2 Syntax and semantics of CHR and its extensions

Where PROLOG represents a top-down, backward chaining computational paradigm, Constraint Handling Rules, CHR, extends with bottom-up, forward chaining computations. Operationally, CHR is defined as rewriting rules over a constraint store, which can be seen as a global resource to be used by a PROLOG program for storing, manipulating and consulting different hypotheses. This resource-oriented understanding of CHR has led to the formulation of a semantics for CHR [12] based on linear logic (that we have not applied here). We introduce firstly CHR, then extend it with disjunctions into $\text{CHR}^{\vee}$ [13], and finally we combine it with PROLOG into a language which we refer to in this paper as PROLOG+CHR.

2.1 Constraint handling rules, CHR

CHR inherits the basic nomenclature of PROLOG, and we refer to the notions of constant and function symbols, predicates, variables, atoms,[1] terms and queries and use initial capital letter for variables, etc. Substitutions, grounding and renaming substitutions are defined as usual.

The predicates in a CHR program are called *constraint predicates*, belonging to disjoint sets of *program specific* ones and a fixed set of *built-in* predicates each having a fixed meaning, including = with its standard meaning of syntactic equality, and `true` and `false`; we assume a theory $\mathcal{B}$ for the built-ins, e.g., $\mathcal{B} \models$ `a` = `a` and $\mathcal{B} \not\models$ `false`. A *constraint* is an atom with a constraint predicate, which may be further classified as *program specific* or *built-in* according to its predicate; in contexts with no ambiguity, 'constraint' may also be used for 'constraint predicate'. A CHR program consists of rules of the following kinds.

Simplification rules: $h_1, \ldots, h_n$ `<=>` $Guard \mid b_1, \ldots, b_m$
Propagation rules: $h_1, \ldots, h_n$ `==>` $Guard \mid b_1, \ldots, b_m$
Simpagation rules: $h_1, \ldots, h_k \setminus h_{k+1}, \ldots h_n$ `<=>` $Guard \mid b_1, \ldots, b_m$

Each h_i is a program specific constraint, each b_i is a program specific or built-in constraint, and the guard G is a possibly empty sequence of built-in ones; we have $n, m \geq 1$ and $n > k \geq 2$. The comma is a sequencing operator which may be interpreted as conjunction or set construction. The constraints to the left of the arrow symbol constitute the *head*[2] of the rule, those to right of the vertical bar the *body*. For simplicity, we require that any variable in the guard of a rule must occur also in its head. An empty guard is interpreted as `true` and may be left out together with the vertical bar.

The rules of a CHR program can be understood operationally as rewrite rules over *states* which are sets of constraints. For the purpose of extending the semantics to PROLOG below, we separate a state into two components as $\langle \Theta, \Sigma \rangle$, where Θ is called the *current query* (also called *unseen* constraints), and Σ the *constraint store* which does not contain equalities, as they are treated in a special way. A state $\langle \emptyset, \Sigma \rangle$ may be identified with Σ, and $\langle \Theta, \emptyset \rangle$ with Θ when no ambiguity occurs. A distinguished state is denoted $\bot$ and understood as falsity or contradiction.

---

[1] There is a slight confusion here. Some PROLOG sources use 'atom' to refer to constants; we use 'atom' in the same way as the literature on mathematical logic, i.e., *predicate*`(`$term_1$, ..., $term_n$`)`, $n \geq 0$.

[2] Primary sources on CHR refer to each c_i, $i = 1..n$ as a head, and thus such rules as multiheaded; complying with our own publications, we let 'head' refer to the whole lefthand side.

A rule can apply if it has an instance so that 1) the head constraints become identical to constraints in the store, and 2) the instantiated guard is satisfied. Simplification rules implement bi-implication by replacing a set of constraints in the store by other, equivalent constraints, while propagation rules implement implication by adding new constraints without removing their 'antecedents'. Simpagation rules are, as their name suggests, a mix of the two others: head constraints indicated before the backslash stay in the store and those after are removed. Equality atoms $s = t$ are executed as in Prolog by applying a most general unifier of s and t to the state. These principles define an abstract derivation relation over states, denoted $\rightsquigarrow$; we refer to [5] for details. A more detailed, operational semantics, which is deterministic and captures the semantics of the implemented CHR systems, can be found in [14].

A *query* Q is given as a conjunction of constraints. A (general) derivation for Q, given a CHR program $\mathcal{C}$, is given as a sequence of the steps above using rules of $\mathcal{C}$,

$$\langle Q, \emptyset \rangle = S_0 \rightsquigarrow S_1 \rightsquigarrow S_2 \cdots .$$

When the derivation is finite and ends with a state S, we write

$$Q \overset{\mathcal{C}}{\rightsquigarrow} S.$$

If $S = \bot$ we say that the derivation has *failed*; otherwise, if no step can be applied to $S = \langle \emptyset, A \rangle = A$, we say that the derivation has *(successfully) terminated* and that A is a set of *answer constraints* for Q. The *answer substitution* σ related to Q and A is defined as the composition of the most general unifiers applied in the derivation of A restricted to the variables of Q; we may write this $Q \overset{\mathcal{C}}{\rightsquigarrow} A, \sigma$.

Example 2.1 The following CHR program is written in the syntax provided by the libraries supplied with SWI and SICStus PROLOG.

```
:- chr_constraint a/0, b/0.
a,b <=> true.
```

The query `?-a` produces the answer constraint set {`a`} whereas `?-a,b` yields $\emptyset$. While this operational behaviour is perfectly sound with respect to the logic semantics to be introduced, this example indicates also the power of CHR to work very explicitly with generating hypotheses and controlling their scope; above, the call to `a` makes the awareness of hypothesis `a` globally available whereas the effect of `b` in the extended query is effectively to erase `a`. □

As opposed to PROLOG, we need for CHR only a single derivation to characterize the set of all answers to a given query. Notice that CHR in practice applies committed choice which means that no alternatives can be obtained backtracking, and a failure in one step means failure of the entire execution.

In practice, CHR applies a multiset semantics and it is often desirable to add additional rules to a CHR program that suppress duplicates. The detailed operational semantics may be employed by the programmer to obtain effects that are otherwise difficult to achieve within an abstract, nondeterministic semantics. In the few cases we employ such tricks below, the procedural details are explained.

A CHR program $\mathcal{C}$ is called *failure-safe* if, for any Q with $\mathcal{C} \models \forall(Q \leftrightarrow \bot)$, any derivation leads to the state $\bot$. A CHR program is called a *constraint solver* if it is failure-safe and any derivation terminates. The stronger property of confluence is not needed for our results and sometimes not even desirable; it means that any derivation for a given query leads to equivalent answers independently of which alternative derivation is chosen; see [5].

Example 2.2 The following CHR program defines a constraint solver.

```
:- chr_constraint here/1, there/1.
here(X), there(X) ==> fail.
```

It returns $\bot$ whenever a query implies that some value v is both `here(`v`)` and `there(`v`)`. For queries without this property and without built-ins, it returns the query as answer. □

Example 2.3 A useful built-in is the `dif/2` predicate provided by SICStus PROLOG, which serves as a lazy non-equality test. Consider the constraint `dif(`s`,`t`)`: if s and t become non-unifiable, the constraint vanishes; when they become identical, a failure is produced; otherwise it remains silent. When an infinity of constant symbols are assumed, any finite set of reduced `dif` constraints is satisfiable, i.e., each of the form `dif(`s`,`t`)` where s and t are unifiable and non-identical. □

The rules of CHR are specific logic formulas written in a notation adapted for computers with implicit quantifiers. The translation is given as follows,

	Propagation rule	Simplification rule
CHR rule:	H`==>`$G \mid B$	H`<=>`$G \mid B$
Logical meaning:	$\forall \bar{x}(G \rightarrow (H \rightarrow \exists \bar{z} B))$	$\forall \bar{x}(G \rightarrow (H \leftrightarrow \exists \bar{z} B))$

where $\bar{x}$ refers to the variables in H and $\bar{z}$ to those in B not overlapping with $\bar{x}$.[3] Here we understand a simpagation rule $H_1 \setminus H_2$`<=>`$G \mid B$ as an abbreviation of the simplification rule H_1, H_2`<=>`$G \mid H_2, B$ and translate it into

[3] Recall that we required, for simplicity only, that any variable in the guard appears in the head as well.

a logical formula accordingly. From [5] we have the following results which are straightforward to show by induction. For simplicity, we formulate correctness conditions for ground queries only.

Theorem 2.1 (Soundness and completeness of CHR derivations)
Let $\mathcal{C}$ be a CHR program, Q a ground query and A an answer constraint set, i.e., $Q \overset{\mathcal{C}}{\leadsto} A$. Then $\mathcal{C}, \mathcal{B} \models \forall(Q \leftrightarrow A)$.

Whenever $\mathcal{C}, \mathcal{B} \models \forall(Q \leftrightarrow A')$ for a ground query Q which has at least one terminated derivation, there is an answer constraint set A such that $\mathcal{C}, \mathcal{B} \models \forall(A \leftrightarrow A')$ and $Q \overset{\mathcal{C}}{\leadsto} A$. □

While soundness is as one may expect, the completeness part is rather weak in the sense that not every constraint store which is a logical consequence of the program will be found.

Example 2.4 Let $\mathcal{C}$ consist of the rules `a<=>b` and `c<=>b`. We have that $\texttt{a} \overset{\mathcal{C}}{\leadsto} \{\texttt{b}\}$ and that $\mathcal{C}, \mathcal{B} \models \forall(\texttt{a} \leftrightarrow \texttt{b})$. On the other hand, it holds that $\mathcal{C}, \mathcal{B} \models \forall(\texttt{a} \leftrightarrow \texttt{c})$, but it is not the case that $\texttt{a} \overset{\mathcal{C}}{\leadsto} \{\texttt{c}\}$. □

This indicates that a CHR programmer needs to be aware of the operational semantics to have the desired answers produced. When a deterministic operational semantics is introduced, this is even more apparent.

The following example shows that the requirement of a finite derivation is necessary in theorem 2.1.

Example 2.5 (From [5]) Consider the program consisting of the single rule `p<=>p`. For the query `p` we have that $\mathcal{C}, \mathcal{B} \models \forall(\texttt{p} \leftrightarrow \texttt{p})$, but there are no answer constraint sets since no derivation terminates. □

For CHR programs of propagation rules only, derivations generate models. The answer constraint set for a query Q is the smallest model of the program which contains Q.

Example 2.6 Consider the following CHR program which extends example 2.2.

```
:- chr_constraint here/1, there/1, distant/2.
here(X), there(X) ==> fail.
here(X), there(Y) ==> distant(X,Y).
```

The query `?- here(me), there(you)` generates as answer the model {`here(me)`, `there(you)`, `distant(me,you)`}, whereas `?- here(you), there(you)` generates $\bot$ meaning that there is no model containing those two atoms. □

For historical and conceptual reasons, we introduce the language CHR$^\vee$ [13] which is an extension of CHR with possible disjunctions in the body, denoted by semicolons. An operational semantics for CHR$^\vee$ is provided by extending the $\rightsquigarrow$ relation with the principle of selecting nondeterministically one of the candidates of a disjunction.

In practice, since disjunctions are interpreted by the underlying Prolog system, the different alternatives are tried out under backtracking. In order to present a completeness property, we define a CHR$^\vee$ *tree* for a query Q and program $\mathcal{C}$ as a tree which has state $\langle Q, \emptyset \rangle$ as root. A node N may have exactly one subtree labeled N' whenever $N \rightsquigarrow N'$, except for disjunctions, where each alternative gives rise to a subtree. A tree is *final* whenever it is finite and no more subtrees can be added; a tree and the query at its root are said to be *failed*, if every branch ends with $\bot$.

The advantage of CHR$^\vee$ over CHR is that it provides a way to explore different possible hypotheses as may be needed in abductive reasoning.

Example 2.7 Consider the following CHR$^\vee$ program in which `q` can be investigated in two alternative ways.

```
:- chr_constraints q/0, a/0, b/0, c/0.
q <=> a ; c.
a, b <=> false.
```

The query `?-q` leads to two possible answer constraint sets {`a`} and {`c`}, whereas `?-b,q` has only one, namely {`b`, `c`}. □

The declarative semantics of CHR$^\vee$ is defined in the same way as for CHR, with semicolon read as $\vee$. The following results generalizes theorem 2.1.

Theorem 2.2 (Soundness and completeness of CHR$^\vee$ derivations) Let $\mathcal{C}$ be a CHR$^\vee$ program, Q a ground query, and assume there is a final CHR$^\vee$ tree which contains the answer constraint sets $A_1, \ldots, A_n$. Then it holds that

$$\mathcal{C}, \mathcal{B} \models \forall(Q \leftrightarrow A_1 \vee \cdots \vee A_n)$$

and

$$\mathcal{C}, \mathcal{B} \models \forall(A_i \rightarrow Q), \text{for all } i,\ 1 \leq i \leq n.$$

Whenever $\mathcal{C}, \mathcal{B} \models \forall(A' \rightarrow Q)$ for a ground query Q which has at least one terminated derivation, there is an answer constraint set A such that $\mathcal{C}, \mathcal{B} \models \forall(A' \rightarrow A)$ and $Q \overset{\mathcal{C}}{\rightsquigarrow} A$. □

Example 2.8 We consider an example from the logic programming folklore. Let `gw` stand for `grass_is_wet`, `r` for `rained_last_night`, and `s` for `sprinkler_was_on`. Consider first the following program $\mathcal{C}_1$ consisting of a simplification rule.

```
:- chr_constraint gw/0, r/0, s/0.
gw <=> r ; s.
```

The query `?- gw` leads to two final constraint sets {`r`} and {`s`}. We confirm the theorem noting that $\mathcal{C}_1 \models \texttt{gw} \leftrightarrow \texttt{r} \vee \texttt{s}$, and that $\mathcal{C}_1 \models \texttt{r} \rightarrow \texttt{gw}$. We notice that if, in the completeness part, we let $Q = \texttt{gw}$ and $A' = \{\texttt{gw}\}$, we can use $A = \{\texttt{r}\}$.

Consider another program $\mathcal{C}_2$ in which the simplification rule is replaced by a propagation rule `gw ==> r ; s`. Here we get the final constraint sets {`gw`, `r`} and {`gw`, `s`}. Again, we confirm the theorem noting that $\mathcal{C}_2 \models \texttt{gw} \leftrightarrow (\texttt{gw} \wedge \texttt{r}) \vee (\texttt{gw} \wedge \texttt{s})$ and that $\mathcal{C}_1 \models \texttt{gw} \wedge \texttt{r} \rightarrow \texttt{gw}$. We notice that if, in the completeness part, we let $Q = \texttt{gw}$ and $A' = \{\texttt{gw}\}$, we can use $A = \{\texttt{gw}, \texttt{r}\}$. □

2.3 PROLOG+CHR

Another way to explore different possible hypotheses is to combine PROLOG with CHR, where the disjunction is provided by alternative clauses for the same PROLOG predicate; we refer to this combined language as PROLOG+CHR. A PROLOG+CHR program has constraint and built-in predicates as for CHR and additionally a set of PROLOG *predicates*, disjoint from the two others; an atom with a PROLOG predicate is called a PROLOG *atom*. A PROLOG+CHR program $\langle \mathcal{P}, \mathcal{C} \rangle$ has two components, a CHR$^\vee$ program $\mathcal{C}$ and a finite set $\mathcal{P}$ of *clauses* of the form

h `:-` b_1`,` $\ldots$`,` b_n, $n \geq 1$

where h is a PROLOG atom called the *head* of the clause, and $b_1, \ldots, b_n$ called the *body* may consist of PROLOG atoms and constraints. (In some of our applications, disjunctions in CHR rule bodies may be useful as a way to have a constraint solver produce alternative solutions, we do not need this in the bodies of PROLOG rules). A clause of the form h`:-true` is called a *fact* and written as h; any other clause is called a *rule*. In PROLOG+CHR we allow also calls to PROLOG predicates in the bodies of CHR rules.

The operational semantics for CHR is extended to PROLOG+CHR in the standard way: when a rule is applied to a PROLOG atom, it is unified with the head of the clause, leading to either failure, i.e., $\bot$, or the addition of the atoms from the clause body to the current query.

A PROLOG+CHR *tree* for a query Q and program $\langle \mathcal{P}, \mathcal{C} \rangle$ is defined analogously to CHR$^\vee$ trees, with alternative subtrees corresponding to the possible clauses that may apply. All other notions are defined as before.

The operational semantics of available PROLOG+CHR systems uses the standard top-down, left-to-right, textual-order computation rule of PROLOG. When a constraint c is encountered and a CHR rule is called, its body B is executed also in the top-down, left-to-right. When this B has executed, and if c has not been consumed by a simplification or simpagation step, the system searches for other CHR rules that may be activated by c, and so forth. In case no PROLOG predicates are called from the CHR rules, this means than when a constraint is called, the CHR rules continue as long as possible and then give back control to the calling PROLOG rule. When, furthermore, the CHR part of a PROLOG+CHR program $\langle \mathcal{P}, \mathcal{C} \rangle$ satisfies the conditions for being a constraint solver (failure-safe and always terminating), the programmer may consider $\mathcal{C}$ as a black-box which tests whether c is compatible with the already accumulated constraints; it may return the updated set of constraints in a 'solved form', which means that execution continues along the given branch, or $\bot$, forcing PROLOG to backtrack.

Example 2.9 The following PROLOG+CHR is written in the syntax provided by SWI and SICStus PROLOG. Constraint predicates are declared as in CHR above and PROLOG predicates such as `q` are given implicitly by their use as usual.

```
constraints a/0, b/0, c/0.
q:- a.
q:- c.
a,b <=> false.
```

This program behaves in the same way as the CHR$^\vee$ program of example 2.7 above, so for example `?-b,q` leads to one answer constraint set $\{\mathtt{b}, \mathtt{c}\}$. □

A general method for rewriting PROLOG programs into CHR$^\vee$ programs was given by [13], which generalizes trivially to PROLOG+CHR; this has been illustrated by examples 2.7 and 2.9.

The advantages of using PROLOG+CHR instead of CHR$^\vee$ are more modular program structures, a natural separation between backward and forward chaining parts, and not least efficiency and acceptance in the logic programming society. Furthermore, it provides access to use all facilities of a fully instrumented PROLOG environment, including Definite Clause Grammars and various high and low level auxiliaries.

There is an important difference in the ways the PROLOG and CHR rules are applied. PROLOG uses unification between the head of a rule and an atom

in the state; this unification may specialize both and also affect other parts of the state. CHR applies a different sort of matching: a specialization or an instance of the rule is made such that the head constraints are found as copies in the state; this does not in itself have any other side affects to the state.

Example 2.10 Consider the following PROLOG+CHR program showing different sorts of rules.

```
:- chr_constraint c1/1, c2/1.
p1(a).
p2(X):- X=a.
c1(X) <=> X=a.
c2(a) <=> true.
```

Queries `?-p1(X)` and `?-p2(X)` yield answer substitution `X=a`. For `?-c1(X)` and `?-c2(X)`, only the first one leads to `X=a` whereas the second returns the query unaltered as answer constraint. □

The logic semantics of a PROLOG program can be described by the so-called Clark completion [15]; see also [16]. While each single clause stands for an implication formula, the set of all clauses in a program with common head predicate are combined into a bi-implication ('only-if').

For a program of PROLOG+CHR, we take as its logic meaning the conjunction of the completion of its PROLOG part and the meaning of its CHR part as given above. We combine standard results about PROLOG with theorem 2.1 into the following.

Theorem 2.3 (Soundness and completeness of PROLOG+CHR derivations) Let $\langle \mathcal{P}, \mathcal{C} \rangle$ be a PROLOG+CHR program, Q a ground query, and assume there is a final PROLOG+CHR tree which contains the answer constraint sets $A_1, \ldots, A_n$. Then it holds that

$$\mathcal{P}, \mathcal{C}, \mathcal{B} \models \forall(Q \leftrightarrow A_1 \vee \cdots \vee A_n)$$

and

$$\mathcal{P}, \mathcal{C}, \mathcal{B} \models \forall(A_i \rightarrow Q), \text{for all } i,\ 1 \leq i \leq n.$$

Whenever $\mathcal{C}, \mathcal{B} \models \forall(A' \rightarrow Q)$ for a ground query Q which has at least one terminated derivation, there is an answer constraint set A such that $\mathcal{C}, \mathcal{B} \models \forall(A' \rightarrow A)$ and $Q \overset{\mathcal{P},\mathcal{C}}{\rightsquigarrow} A$. □

If we furthermore require that the CHR part of a PROLOG+CHR program is a constraint solver, we can remove the requirement in the completeness part about existence of a terminated derivation.

2.4 Additional facilities in current PROLOG+CHR *systems*

Here we mention briefly facilities of current PROLOG and CHR systems that are useful for hypothesis-based reasoning and that are applied in the remainder of this paper.

2.4.1 Syntactic and semantic extensibility

PROLOG's operator definitions combined with the so-called term expansion facilities available in, among others, SICSTUS PROLOG are effective tools for the programmer to provide a syntax and implementation for variations of the logic programming paradigm.

We illustrate the principle here by an example, which should be sufficient for understanding the applications later in this paper. For details and variations between the PROLOG versions, we refer to [10,11].

Example 2.11 The CHRG and HYPROLOG systems [17,18] both include a `where` notation which may help to make complicated programs more readable. So, for example, the clause `p(X):- r(X,Y), z(Y,17), q(X)` may be written alternatively in the following way,

```
p(X):- Test, q(X)
where Test = ( r(X,Y), z(Y,17) ).
```

The `where` device is defined as an operator with suitable precedence and associativity in the usual PROLOG way, and its 'meaning', i.e., replacing the variable `Test` by the term given after the `where` symbol, is defined by adding a suitable clause to the system predicate `term_expansion`; details are straightforward and omitted. □

2.4.2 Dynamic assertion and retraction of PROLOG *clauses*

Using PROLOG's familiar `asserta`, `assertz`, and `retract` primitives, it is possible to accumulate and manipulate information in a global state during the execution of a PROLOG program. In some cases, e.g., when constraints are known always to be ground, CHR constraints can be rewritten through these facilities, but the CHR rules need to be rewritten in an awkward way. We consider this program style as obsolete, but mention the possibility as a target for automatic or systematic, manual translation when reasons of efficiency may make it relevant. In section 7.3 below, we show details and compare efficiency for selected examples.

2.4.3 Low-level CHR *features*

The different versions of CHR have different collections of low-level tools, some of which are useful for implementation of new facilities for hypothesis-based reasoning. Some of these may be hard to relate to a declarative semantics, but applied in a structured way and hidden well inside abstractions, they can provide a logical behaviour that goes beyond what is expressed naturally in plain CHR. Such facilities may include explicit inspection of the constraint store, including adding and deleting constraints bypassing CHR's normal mode of working. Finally, we mention the so-called `passive` declarations which can be added to a CHR rule, and which are useful for a more detailed control of when rules are applied.

Example 2.12 Consider the following rule which is a simplification rule extended with a `passive` declaration.

```
a(X), b(X)#Id <=> c(x) pragma passive(Id).
```

This rule cannot fire due to a call to `b/1`, but a call to `a/1` initiates a search for a possible companion `b/1` constraint. So the query `?-a(1),b(1)` returns the constraints unaltered as answer, whereas `?-b(1),a(1)` returns {`c(1)`}. □

Passive declarations can be used for optimization alone or for changing the logic of the program (analogously to the way '`!`' is used by PROLOG programmers); [19] analyzes the different usages of passive declarations.

3 Abductive logic programming with CHR$^\vee$ and PROLOG+CHR

3.1 Abductive logic programs

Here we give a definition of Abductive Logic Programming adapted from [9]. An abductive logic program [2] is a triplet $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ where $\mathcal{P}$ is a logic program, $\mathcal{A}$ a set of *abducible* predicates that do not occur in the head of any clause of $\mathcal{P}$, and $\mathcal{IC}$ a set of integrity constraints assumed to be consistent. A (not necessarily ground) atom with an abducible predicate is called an *abducible (atom)*. We assume additionally that $\mathcal{P}$ and $\mathcal{IC}$ can refer to a set of *built-in* predicates that have a fixed meaning given by the theory $\mathcal{B}$. Given abducible predicates $\mathcal{A}$ and integrity constraints IC, we define for a set of abducibles and built-in atoms $A \cup I$, a *consistent ground instance* to be a common ground instance $A' \cup I'$ of $A \cup I$ so that

- $\mathcal{B} \models I'$ (the instance of built-ins is satisfied)
- $\mathcal{B} \cup A' \models \mathcal{IC}$ (the instance of abducibles respects the integrity constraints)

For simplicity and without loss of generality, we consider only ground queries; an *abductive explanation* for a query Q and abductive logic program $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ is a set of abducibles and built-in atoms $A \cup I$ such that

- $A \cup I$ has at least one consistent ground instance $A' \cup I'$,
- for any such $A' \cup I'$, we have $\mathcal{P} \cup A' \models Q$.

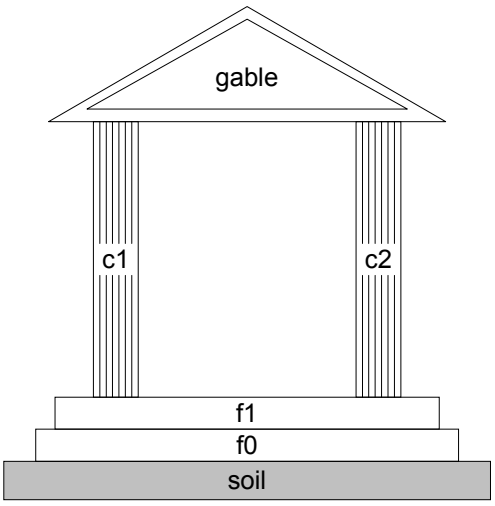
3.2 PROLOG+CHR *programs as abductive logic programs*

The following theorem shows the relationship between PROLOG+CHR programs and a specific form of abductive logic programs. The proof follows immediately from theorem 2.3 and the classical deduction theorem, which states that $\Gamma, \gamma \models \phi$ iff $\Gamma \models \gamma \rightarrow \phi$; see, e.g., [20] or another good book on mathematical logic.

Theorem 3.1 Let $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$ be an abductive logic program that coincides with a PROLOG+CHR program $\langle \mathcal{P}, \mathcal{IC} \rangle$ with $\mathcal{A}$ being the set of program defined constraints, and assume that $\mathcal{IC}$ satisfies the conditions for being a constraint solver. Then any answer constraint set for a ground query Q to $\langle \mathcal{P}, \mathcal{IC} \rangle$ is an abductive explanation for Q in the abductive program $\langle \mathcal{P}, \mathcal{A}, \mathcal{IC} \rangle$. For any abductive answer A' for a ground query Q, there is an answer constraint set A (for Q in $\langle \mathcal{P}, \mathcal{IC} \rangle$) such that $\mathcal{IC}, \mathcal{B} \models \forall(A' \rightarrow A)$. □

A similar theorem holds for CHR$^\vee$ with the PROLOG part of the abductive logic program rewritten, via its completion, to a set of simplification rules.

Example 3.1 (Adapted from [18]) We consider a planning problem in construction work, concerned with the building of a facade of a classical temple. The architect's sketch is as follows, naming the different parts and showing their position in the desired construction.



The following program expresses the problem as an abductive logic program written in PROLOG+CHR. We provide the entire source file, including the directive to include the CHR library, so the text can be copied and run directly in SICSTUS PROLOG.

```
:- use_module(library(chr)).
:- chr_constraint put_in_place/2. % abducibles put_in_place(Object,Time)

%%% A part cannot be placed earlier that a part that supports it:
put_in_place(P0,Time0), put_in_place(P1,Time1) ==>
  supports(P0,P1), Time0 > Time1 | fail.

build:-  put_in_place(soil,0), parts(Parts), build(Parts,1).

build([],_).

build(Parts,Time):-
   takePart(P,Parts,RestParts),
   put_in_place(P,Time),
   Time1 is Time+1,
   build(RestParts,Time1).

parts([gable,c1,c2,f0,f1]).
supports(soil,f0).
supports(f0,f1).
supports(f1,c1).
supports(f1,c2).
supports(c1,gable).
supports(c2,gable).

takePart(X,List0,List1):-
    append(LeftRest,[X|RightRest],List0),
    append(LeftRest, RightRest,List1).
```

The following dialogue shows the query and an example of an abductive answer expressing a plan, i.e., a specification of which part of the construction should be placed at which times.

```
?- build.
put_in_place(gable,5),
put_in_place(c2,4),
put_in_place(c1,3),
put_in_place(f1,2),
put_in_place(f0,1),
put_in_place(soil,0) ?
```

□

The teaching note [21] provides further examples, including diagnosis of faulty logical circuits, where small adjustments of the integrity constraints can tune the program to work under different assumptions of periodic faults, consistent faults, and that observed correct behaviour always is produced in a correct way, i.e., excluding the option that different faults compensate for each other.

Example 3.2 The dining philosophers problem was introduced in [22] as a prototype problem for process synchronisation and also considered in the literature on abduction. We use here a formulation inspired by [23]. Consider (here) five philosophers sitting at a round table with one chopstick placed between each two philosophers. In order to eat, a philosopher needs two chopsticks which he can take from the table and but back afterwards. Clearly, no two neigbouring philosophers can eat at the same time. We use the following constraint solver to indicate the overall conditions.

```
:- chr_constraint chop_in_use/3, % chop_in_use(ChopId, PhiId, Time)
                  chop_free/2,   % chop_free(ChopId, Time)
                  eating/3.      % record who ate when:
                                 % eating(PhiId, Start, End)

chop_in_use(C,Ph,Tx), chop_free(C,Ty)
                               ==> true | (Tx=Ty ; dif(Tx,Ty) ).
chop_in_use(C,Phx,T), chop_in_use(C,Phy,T) ==> Phx=Phy.
```

Th first integrity constraint allows to take a chopstick which is free at time t to be used by a philosopher starting from t; the second one states that only one philosopher can use a given chopstick at any given time.

The following PROLOG facts describe the positioning around the table and a simple implementation of time, indicating possible successive 'eating intervals'.

```
compute_needed_chops(ph1,c1,c2).          eat(t0,t1).
compute_needed_chops(ph2,c2,c3).          eat(t1,t2).
compute_needed_chops(ph3,c3,c4).          eat(t2,t3).
compute_needed_chops(ph4,c4,c5).          eat(t3,t4).
compute_needed_chops(ph5,c5,c1).          eat(t4,t5).
                                          eat(t5,t6).
```

The sequence of actions performed by each philosopher is described by the following PROLOG rule.

```
phil(P):-
    compute_needed_chops(P,C1,C2),
    chop_in_use(C1,P,T1), chop_in_use(C2,P,T1),
    eat(T1,T2), eating(P,T1,T2),
    chop_free(C1,T2), chop_free(C2,T2).
```

For simplicity, it is assumed that each philosopher wants to eat just once. Now the following query defines the initial setup at time `t0` and poses the philosophers' desire to eat.

```
?- chop_free(c1,t0), chop_free(c2,t0), chop_free(c3,t0),
   chop_free(c4,t0), chop_free(c5,t0),
   phil(ph1), phil(ph2), phil(ph3), phil(ph4), phil(ph5).
```

A number of solutions are produced on backtracking; here we show the first and fifth.

```
eating(ph5,t4,t5),            eating(ph5,t1,t2),
eating(ph4,t3,t4),            eating(ph4,t0,t1),
eating(ph3,t2,t3),            eating(ph3,t2,t3),
eating(ph2,t1,t2),            eating(ph2,t1,t2),
eating(ph1,t0,t1)             eating(ph1,t0,t1)
```

Adding more integrity constraints, we can ensure optimal solutions (as the indicated fifth) in the sense that chopsticks are taken up if possible at any given time. □

The following example shows linguistic discourse analysis by means of abduction.

Example 3.3 (Adapted from [24]) Abduction is an appreciated metaphor for discourse analysis from natural language texts; see, e.g., [25]. CHR works smoothly together with PROLOG's Definite Clause Grammar notation and, although this is by no means a profound discovery, it provides great advantages in at least the teaching of computational linguistics. Consider the following discourse.

> Garfield eats Mickey, Tom eats Jerry, Jerry was a mouse, Tom is a cat, Mickey was a mouse.

A discourse analysis may reveal to which categories the mentioned characters belong and which categories are food items for which others. A particularly interesting question is to which category Garfield belongs as this is not mentioned explicitly. The following combination of CHR and grammar rules implements the necessary parsing and abductive reasoning framework to do the job. Notice that the CHR rules, i.e., the integrity constraints, are simpagation rules; this removes the duplicate constraints that otherwise would arise with a propagation rule after the unification in the body. The integrity constraints indicate that the category of a given character is unique, and for the sake of this example it is assumed that a given category is the food item for at most one other category.

```
:- use_module(library(chr)).
:- chr_constraint categ_of/2, food_for/2.

categ_of(N,C1) \ categ_of(N,C2) <=> C1=C2.
food_for(C,C1) \ food_for(C,C2) <=> C1=C2.

sentences --> [] ; sentence, sentences.
sentence --> name(N), ([is];[was]), category(C), {categ_of(N,C)}.
sentence --> name(N1), [eats], name(N2),
             {categ_of(N1,C1), categ_of(N2,C2), food_for(C2,C1)}.
name(N) --> [N].
category(C) --> [a], noun(C).
noun(N) --> [N].
```

The analysis of the discourse above yields an abductive answer corresponding to a knowledge base learned from that discourse, which includes among others the abducible `categ_of(garfield,cat)`. A detailed trace shows that the first sentence produces

```
categ_of(garfield,X1), categ_of(mickey,X2), food_for(X1,X2)
```

where the `X`s are currently uninstantiated variables. As the subsequent sentences are analyzed and more abducibles are added to the constraints, the integrity constraints will graduately fire so that the final sentence leads to a unification of variable `X1` with the value `cat`. □

3.3 Abductive logic programming with constraints

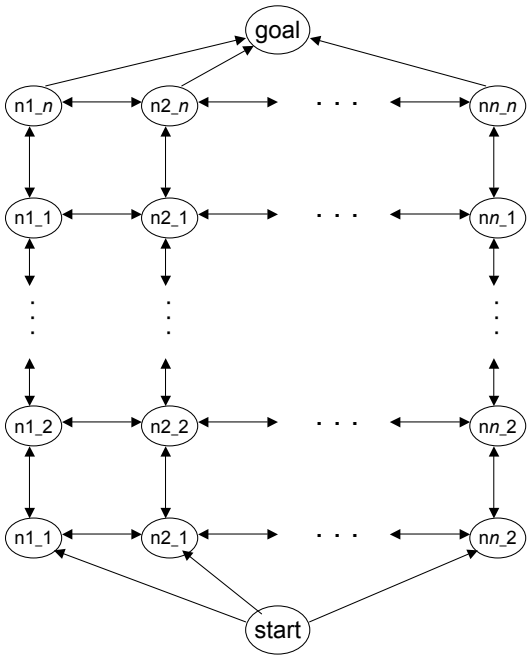
Application of existing constraint solvers may provide effective ways of reducing the search space in abductive reasoning and may also help in producing more concise programs. We refer to such constraints as *external constraints,* and the applied constraint solver as being *external.*[4] Systems for abductive logic programming, which incorporate specific constraint solvers such as finite domain solvers or CLP(R), have been developed [26–28].

Any constraint solver provided by the given PROLOG system can be used in an abductive logic program written in PROLOG+CHR. It works correctly without any special care from the programmer, since each constraint solver applies it own constraint store, which becomes part of the total execution state. When a

[4] This completes the slightly confusing overloading of the term "constraint". Constraints may now refer to *CHR constraints*, taking the role of abducibles, *integrity constraints* which are CHR rules, and now *external* ones which are PROLOG atoms whose meaning is provided by the PROLOG system or one of its libraries.

PROLOG program takes care of the overall control, each constraint solver reacts each time one of "its" constraints are called, performs possible transformations in its own store, producing either a failure or a signal to the PROLOG engine to continue with the next subgoal. However, this holds only when we do not use PROLOG's or other kinds of negation as failure, which then would require an explicit manipulation of the constraint representation. We show here an example which uses a CLP(Q) constraint solver.

Example 3.4 We consider the construction of paths in a graph, whose lengths are subject to restrictions. This example involves nonground abducibles and, for the PROLOG+CHR implementation, the use of an external constraint solver, which here is SICSTUS' `library(clpq)` [29,10].



Nodes are labelled, `start`, `goal`, `n`i_i. Edges, except in the top and bottom, are bidirectional and assigned a random cost. The task is to find three different paths, that we call the red, green and blue path, and which must be disjoint. Furthermore, the collected cost for the red paths must be larger than the one for the green path, successively larger than for the blue path.

Path costs can be calculated incrementally (which is the optimal way), but in order to have the external constraint solver do interesting work, we set up constraints when generating each path (from the top and downwards), so the total and intermediates costs are not known before the path reaches node `start`. However, the program may indirectly compare the costs of paths generated from the cost of the edges already involved and the knowledge the the unknown costs are positive. The graph is given by program facts as follows, here shown for the instance $n = 5$.

```
node(start).
edge(start,n1_1,53).
...
node(n1_1).
edge(n1_1,n2_1,18).
edge(n2_1,n1_1,57).
edge(n1_1,n1_2,89).
edge(n1_2,n1_1,83).
...
node(n5_5).
edge(n5_5,goal,16).
node(goal).
```

The remaining program clauses are as follows, with `path_edge` being the abducible predicate. Code inside the curly brackets is managed by CLP(Q).

```
problem:-
  {RedCost > GreenCost, GreenCost > BlueCost},
  path(red,RedCost), path(green,GreenCost), path(blue,BlueCost).

path(Id,Cost):- path(Id,goal,Cost).
path(_,start,StartCost):- {StartCost=0}.
path(Id,N2,C):- edge(N1,N2,C2), {C = C1+C2, C1>=0},
   path_edge(Id,N1,N2,C), path(Id,N1,C1).
```

Integrity constraints to prevent loops in paths and ensure disjointness are as follows. [5]

```
path_edge(Id,_,N,_), path_edge(Id,_,N,_) ==> fail.
path_edge(Id1,_,N,_), path_edge(Id2,_,N,_) ==>
     Id1 \= Id2, N\=goal | fail.
```

The query `?- problem` produces the following set of abducibles; `path_edge` is abbreviated `pe` and colours to single letters.

```
                     pe(g,n2_5,goal,264)
pe(r,n1_5,goal,287)  pe(g,n2_4,n2_5,193)  pe(b,n4_5,goal,230)
pe(r,n1_4,n1_5,205)  pe(g,n2_3,n2_4,116)  pe(b,n4_4,n4_5,221)
pe(r,n1_3,n1_4,203)  pe(g,n2_2,n2_3,97)   pe(b,n4_3,n4_4,128)
pe(r,n1_2,n1_3,156)  pe(g,n3_2,n2_2,31)   pe(b,n4_2,n4_3,108)
pe(r,n1_1,n1_2,142)  pe(g,n3_1,n3_2,13)   pe(b,n4_1,n4_2,65)
pe(r,start,n1_1,53)  pe(g,start,n3_1,7)   pe(b,start,n4_1,45)
```

□

[5] Current implementations of CHR will never apply a rule such as the first integrity constraint in the path example with the two head literals matching the same constraint. Other systems may need to test that the start nodes are different.

Several other systems for abductive logic programming include the principle that we call compaction, which that, whenever a new abducible atom appears during the execution, the system tries to unify it with existing abducibles if possible. The aim of this is to produce minimal explanations measured in the number of literals. As we have argued elsewhere [9], it is not always desirable to apply this principle.

Example 3.5 Consider the following example of an abducible explanation which indicates that some events `e1` and `e2` have taken place at some locations and points in time.

```
event(e1,P1,T1), place(P1), time(T1),
event(e2,P2,T2), place(P2), time(T2)
```

According to the compaction principle, the place and time constraints can be unified, and the explanation shrinks from 6 to 4 abducibles. However, this adds the extra commitment stating that the two events took place at the same place and the same time, which the problem or its specification very likely may not account for. □

The implementation of abductive logic programs as PROLOG+CHR programs described here does not provide compaction. However, if compaction is desired for a given predicate, it can be implemented by adding a propagation rule. For example 3.5 above, the following rules will equate places and time points as much as possible without violating other integrity constraints.

```
place(P1), place(P2) ==> true | (P1=P2 ; dif(X,Y)).
time(T1), time(T2) ==> true | (T1=T2 ; dif(X,Y)).
```

The `dif` constraint is a SICSTUS buit-in explained in example 2.3 above. (It is necessary to include the `true` guard due to a subtlety concerned with the precedence of the applied operators.) Notice that due to the multiset semantics in current CHR systems, compaction rules may lead to duplicate constraints in the store which may not be desirable; they can be removed, if wanted, with rules such as `place(P)\place(P)<=>true` which will be triggered by the unification and undone on backtracking.

Explicit negation is a form of negation in logic programming based on the axiom $p \wedge \neg p \rightarrow$ `false`. We can implement this with a propagation rule; for an abducible `p` (with, say two arguments), we may let `p_` be a predicate that represents ¬`p`, and add the following rule.

```
p(X,Y), p_(X,Y) ==> false.
```

Example 3.6 Consider the following program, extended with compaction and explicit negation rules.

```
past:- a(1), b(2), b_(1).
obs(X):- a(X),b(X).
```

The query `?-past,obs(X)` produces two answers on backtracking, first `X=2`, secondly a set of constraints on `X`, {`dif(X,1)`, `dif(X,2)`, `a(X)`, `b(X)`}. □

3.5 Providing a specialized syntax for abductive logic programming

The HYPROLOG system [9,18] adds a thin layer of syntactic sugar on top of SICSTUS PROLOG. Abducibles are declared by the syntax illustrated as follows.

```
abducibles p/2, q/1.
```

Using term expansion facilities (section 2.4.1), this is immediately translated into declarations of CHR constraints[6] for the mentioned predicates and their negations, plus the rules that implement explicit negation.

```
:- chr_constraint p/2, p_/2, q/1, q_/1.
p(X,Y), p_(X,Y) ==> false.
q(X), q_(X) ==> false.
```

Compaction is declared by a directive such as

```
compaction q/1.
```

which translates into a rule as follows.

```
q(X), q(Y) ==> true | (X=Y ; dif(X,Y)).
```

In fact, HYPROLOG produces a slightly different rule which applies low-level CHR facilities to remove duplicate constraints in a backtrackable way with no additional rules of the kind indicated above in section 3.4.

HYPROLOG supports other sorts of hypotheses-bases reasoning that are explained later.

[6] HYPROLOG is implemented in SICSTUS PROLOG 3; the syntax shown here is valid only in SICSTUS 4, which we use for consistency throughout this paper.

4 Assumptions in logic programming

We consider here assumptions in logic programming in the sense of [30]; we use a revised syntax introduced in [9]. Assumptions are related to abduction in the sense that they represent hypotheses in a global state that are generated and can be employed during the execution of a program.

As opposed to abduction, the meaning of these assumptions depends on the sequential execution of a PROLOG program, such that the availability of certain hypotheses may be delimited by points in time according to the execution history. A continuation semantics for PROLOG with assumptions is given by [9]; here we will do with an informal presentation.

Given a set of *assumption symbols*, which is a set of PROLOG function symbols, an *assumption* can be any term whose top symbol is an assumption symbol. The following operators that operate on assumptions can be applied in the body of PROLOG rules.

`+`A	Assert linear assumption A for subsequent proof steps. Linear means "can be used once".
`*`A	Assert intuitionistic assumption A for subsequent proof steps. Intuitionistic means "can be used any number of times".
`-`A	Expectation: consume/apply existing intuitionistic assumption in the state which unifies with A.
`=+`A, `=*`A, `=-`A	Timeless versions of the above, meaning that order of assertion of assumptions and their application or consumption can be arbitrary.

When different hypothesis are available for `-`A and `=-`A, they may all be tried under backtracking.

Example 4.1 Assuming that `h` is an assumption symbol, the following query,

```
?- +h(1), -h(X), +h(2), +h(3), -h(Y).
```

yields two possible answers, `X=1`, `Y=3`, with assumption `+(h(2))` remaining in the state, and `X=1`, `Y=2`, with `+(h(3))` remaining in the state. □

It seems quite obvious that it should be possible to implement these assumption operators in CHR using the constraint store for maintaining the collection of assumptions. However, the backtracking among alternative assumptions, represented as constraints, does not fit with CHR's commited choice. This problem can be solved, e.g., by keeping all available assumptions in a list stored as a single constraint. We exemplify here the implementation of `+` and `-`; the full collection of operators can be implemented in a similar way.

Example 4.2 Linear assumptions and expectations can be implemented by the following lines of code.

```
:- chr_constraint (-)/1, (+)/1, assump_list/1.
+A, assump_list(L) <=> assump_list([A|L]).
+A <=> assump_list([A]).
-E, assump_list(L) <=> member(A,L,LRest), assump_list(LRest), A=E.
```

The predicate `member/3` takes out an element of a list and returns the list of the remaining elements; it generates all possible members on backtracking. In provides the behaviour shown in example 4.1 above. □

The HYPROLOG system [9,18], whose implementation of abduction was outlined in section 3.5 above includes also an implementation of assumptions with the full collection of operators shown above. Assumption symbols are declared explicitly, exemplified as follows.

```
assumptions h/1.
timeless_assumptions n/2.
```

The system translates these into declarations of constraints specialized for efficiency for each assumption symbol,`'+h'/1`, `'*h'/1`, `'-h'/1` and `'=+n'/2`, `'=*n'/2`, `'=-n'/2`, and specialized rules similar to those shown in example 4.2. [7]

Assumptions in logic programming have been used in language analysis to model dependencies across tree structures such as pronoun resolution and coordination sentences. The following example is adapted from [9]; see also [18] for more elaborate linguistic examples.

Example 4.3 Consider discourses consisting of sentences such as "Peter likes Mary. She likes him." The following program which runs under the HYPROLOG system, is written using PROLOG's grammar notation and uses abducibles to record the facts embedded in each sentence and assumptions in order to resolve pronouns.

```
assumptions acting/2.
abducibles fact/3.

sentence -->  np(A,_), verb(V), np(B,_), {fact(A,V,B)}.
sentences --> [] ; sentence(S1),sentences(S2).
np(X,Gender) --> name(X,Gender), {*acting(X,Gender)}.
```

[7] The currently available HYPROLOG system, version 0.0, applies low-level CHR primitives available in SICSTUS PROLOG 3 only, but the cleaner implementation shown in example 4.2 could have been used instead.

```
name(peter,masc) --> [peter].
...
np(X,Gender) --> {-acting(X,Gender)}, pronoun(Gender).
pronoun(fem) --> [her].
...
verb(like) --> [likes].
...
```

The rule for `np(···) --> name(···), ...` produces an assumption `*acting(X, Gender)` to indicate that the given name is available for any future pronoun; when a pronoun is encountered, an expectation `-acting(X,Gender)` is issued which then, can be unified with the possible assumptions in the state. In the sample discourse above, `*acting(peter,masc)` and `*acting(mary,fem)` are generated for the first sentence, so that the expectation `*acting(X,fem)` generated for "She" will unify this `X` with `mary`. Thus the following abducibles are generated for the sample discourse, `fact(peter,like,mary)`, `fact(mary, like,peter)`. □

Assumptions have also be used for simulation of resource scheduling; see [18] for an example.

5 An example of hypotheses with priority and scope

CHR lacks facilities for a detailed priority among rules, and hence also for the applications we have shown so far of hypothesis-based reasoning. As the following example indicates, it is possible to assign priority numbers to different hypotheses, and then select the one with the highest number.

Example 5.1 The application of CHR for analysis of use case descriptions for object oriented systems development and converting them into UML diagrams has been considered in [31,32]. The following rule resolves pronoun reference according to a heuristic of taking the most recent, relevant referent.[8] It is adapted from [31] with slightly enhanced readability using the `where` notation introduced in example 2.11 above. Constraints `referent/4` and `expect_referent/3` are used similarly to the assumptions and expectations described in section 4 above; arguments `No` can be `plural` or `singular`,

[8] It should be stressed that pronoun resolution, and anaphora resolution in general, is one of the most difficult tasks in computational linguistics; the interested reader is referred to [33] for an insightful and entertaining exposition of some of the phenomena that should be taken into account. The simplistic heuristic chosen in the present example is only acceptable here as the texts in question are expected to be highly stereotypical.

`G` stands for gender, and `Id` and `X` identify given referents, e.g., `peter`. Referents are numbered by the sentence number (in the given discourse) from which priorities are calculated. When, say, "Peter" is mentioned in sentence no. 7, a constraint `referent(sing,masc,peter,7)` is emitted, and an occurrence of "him" in sentence no. 10 gives rise to `expect_referent(sing, masc,X)`; the following rule attempts to bind `X` to the suitable value.

```
sentence_no(Now), referent(No,G,Id,T) \ expect_referent(No,G,X) <=>
   T < Now, \+ A_more_recent_referent_possible
 |  ( An_equally_good_referent_possible
       -> X = error:pronoun:ambiguous(No,G,Now)
     ; X=Id )
where
   A_more_recent_referent_possible = (
      ( find_constraint( referent(No,G,_,TMoreRecent) ),
        T < TMoreRecent, TMoreRecent\==Now) ),
   An_equally_good_referent_possible = (
      find_constraint( referent(No,G,Id1,T) ), Id1\=Id ).
```

The current sentence number is kept in the constraint `sentence_no(Now)`. A given referent can only be chosen if it is the most recent one with the right feature values. The test in the body leads to an error code in case there are two equally and most recent possibilities; this means that the heuristic gives up in cases like "Peter and Paul ... . He ..." The predicate `find_constraint` is a primitive available in SWI and SICStus PROLOG which unifies its argument with an existing constraint if possible (with different syntax and different degree of documentation in the different PROLOG versions). □

This example indicates also an obvious need for a higher-level notation to express such priorities and selection criteria. Proposals have been made for expressing priorities in logic program, which may be used in the present context, e.g. [34–36]. We mention also recent work suggesting priorities in CHR [37].

6 An example of nonmonotonic reasoning in CHR

The paper [38] describes an implementation in CHR of a language for agent-oriented programming language called Global Abduction introduced in [39,40]. The idea in Global Abduction is basically that one or more agents work on solving a problem and when doing this, they maintain a common knowledge base reflecting the state of a changing world. Each branch of the computation keeps a record R of which beliefs it has applied from the common knowledge base CB in its proof steps; as soon as R becomes inconsistent with CB due to a nonmonotonic update of CB, it delays (perhaps indefinitely) until CB

and R become consistent; then it may take up its computations. In this way knowledge learned about the world in one branch becomes available to other branches, and 'old', partial solutions to the given problem may be reused when they fit the current word.

The global knowledge base is represented by constraints `(+)/1` and `(-)/1` that hold information known to be true or false, respectively; notice that this usage of plus and minus is totally unrelated to the use of these symbols for the assumptions in logic programs considered in section 4.

Nonmonotonicity means that when a new belief, say `+a`, is added to the database, it overrides a possible old belief about the opposite, i.e., `-a`. The management of the global knowledge base is given by the following rules, the two last ones removing duplicates in a trivial way.

```
constraints (+)/1, (-)/1.

+X \ -X#Old <=> true pragma passive(Old).
-X \ +X#Old <=> true pragma passive(Old).
+X \ +X <=> true.
-X \ -X <=> true.
```

If the knowledge base contains `-a` and process calls `+a`, then the first rule, which is a simpagation, fires and thereby removes `-a` and replaces it by `+a`; the passive declarations ensures that it is always the currently called constraint that takes precedence.

Since Global Abduction delays a process that does not fit the global knowledge base and may restart it when this is not the case anymore, there is no need for complicated updates of each process state. The necessary delay and restart mechanisms are described in [38].

7 Evaluation and comparison with related work

The language of PROLOG+CHR (or CHR$^\vee$ for that matter) is purely deductive in the sense that it produces only answers that are logical consequences of the program at hand. Referring to the deduction theorem, we twisted the correctness statement (theorem 2.3) into a specification of abductive problem solving, and thereby obtaining an implementation. This principle is also inherent in a paper by Console et al [41] from 1991, that also explained abductive reasoning in terms of deduction; however, they did not suggest an underlying implementation platform with the desired deductive capabilities. The idea is extended into a procedure for abduction described as an abstract algorithm by Fung and

Kowalski in 1997 [42], which inspired several implemented systems. See [1–4] for overview and detailed references to a plethora of different approaches and implementations. The main point of our work in comparison, indicates that we have identified a match of technology invented and implemented efficiently by others, namely PROLOG+CHR, and applied it in the approach to abduction by deduction.

The price to be paid is the lack of any interesting use of negation, only the very limited form of so-called explicit negation can be used; we used notation `a_` to represent negation of `a`. For example, with a rule `a(X), b(X) ==> false` we do not automatically deduce `b_(1)` from `a(1)`; if we want this, the rule should be rewritten as `a(X) ==> b_(X)` and `b(X) ==> a_(X)` assuming rules for explicit negation as described in section 3.4. However, for rules that are more complicated that this, such a rewriting becomes very difficult.

As another example of hypothesis-based reasoning in CHR, [43] shows how description logic can be expressed in CHR. The DEMOII system [44,45] uses a combination of CHR and PROLOG to implement a powerful reasoning system based on a reversible version of the classical metainterpreter `demo(`*program*`,` *query*`)` for logic programs with the meaning that the query is provable in the program; reversibility means the system will try to instantiate metavariables in the program argument standing for unknown program components in order to make the query provable. Applications includes abduction, default logic and simplified induction problems. The paper [46] analyzes PROLOG and CHR as general metaprogramming languages.

CHR has also be applied in different ways for language analysis. We have shown above, how hypothesis-based reasoning such as abduction and assumption implemented in CHR works smoothly together with PROLOG's grammar notation; [8] has related this to a possible worlds semantics and linguistic theories about context comprehension. The CHR GRAMMAR system [17,24], or CHRG for short, translates a grammar notation into CHR rules that parse a text bottom-up, and which also interacts with various sorts of hypotheses. Other applications of CHR for language processing that are not commented on here may be found in [7,47–52].

Finally we refer to the website for Constraint Handling Rules which maintains references to all known papers and applications of CHR [53].

7.1 Comparison with other selected systems

Here we describe a number of comparable approaches for which implementations are available; a table of benchmarks is provided for some of them, section 7.3 below.

- Implementation in PROLOG+CHR as described in section 3.2 and also with an external CLP(Q) constraint solvers.
- PROLOG with assert-retract, section 2.4.2, which for programs with guaranteed ground abducibles may provide high efficiency.
- A straightforward meta-interpreter implementation of abductive logic programs with ground abducibles. We adapted a version described in [54,55].
- Abductive logic programming systems extended with finite domain constraints. We consider CIFF [28] and the ASYSTEM [26]. Both implementations involve meta-interpretation, and both have concentrated on efficient employment of SICSTUS PROLOG's finite domain solver. This solver basically stores the constraints without any reduction or attempts to detect failures, until the user program explicitly calls an instantiation phase, called labeling. CIFF applies a strategy of testing if a labeling is possible from time to time (i.e., not for every computation step) to ensure consistency of the collected finite domain constraints; ASYSTEM performs incremental consistency checking and applies a strategy for selecting the next subgoal which looks at the current domain for constrained variables.
- For comparison with the above, we test implementations in PROLOG using the finite domain solver without involving any abductive machinery or special strategy for tuning the performance of the constraint solver.
- Answer set programming (ASP) systems, which compute stable models bottom-up, and in which some abductive problems with integrity constraints can be expressed. DLV [56] is known to be an efficient, state-of-the-art ASP system, but we made no tests as DLV's lack of functions symbols and constraint solving made it unfit with the test suite introduced below.

  We used instead the SMODELS system [57] as it handles function symbols. In the processing of a query, this system applies first a grounding preprocessing phase which produces representations of all possible ground instance of the clauses, including integrity constraints, of the program. It is concluded below, that this is prohibitive for a large class of abductive problems, as the number of these instances grows exponentially with the number of variables in each clause. Ideas for combining ASP with constraint solving are emerging [58,59] but we have not tested any running systems.
- BINPROLOG [60] includes a hardwired implementation of assumptions; an earlier test [9] indicates that assumptions executes about 3 times faster in BINPROLOG than in PROLOG+CHR (not shown in the test table below).

7.2 A suite of test programs

Example 7.1 (Sample program: db update) Database update through views are typical examples used for demonstrating abductive systems. We use an example with three tables and a view predicate `w/3` which joins all tables. Each table has a key constraints, shown as propagation rules below.

```
:- chr_constraint abd_p/4, abd_q/4, abd_r/3.
w(E,F):- pp(A,B,C,D), qq(C,D,E,F), rr(A,E,F).
pp(A,B,C,D):- p(A,B,C,D).
pp(A,B,C,D):- abd_p(A,B,C,D).

qq(C,D,E,F):- q(C,D,E,F).
qq(C,D,E,F):- abd_q(C,D,E,F).

rr(A,E,F):- r(A,E,F).
rr(A,E,F):- abd_r(A,E,F).

abd_p(A,B,_,_) ==> \+ p(A,B,_,_).  % A,B key
abd_q(C,D,_,_) ==> \+ q(C,D,_,_).  % C,D key
abd_r(A,E,_) ==>   \+ r(A,E,_).    % A,E key

p(a1,b1,c1,d1).    ...   p(a100,b100,c100,d100).
q(c1,d1,e1,f1).    ...   q(c100,d100,e100,f100).
r(a1,e1,f1).       ...   r(a99,e99,f99).
```

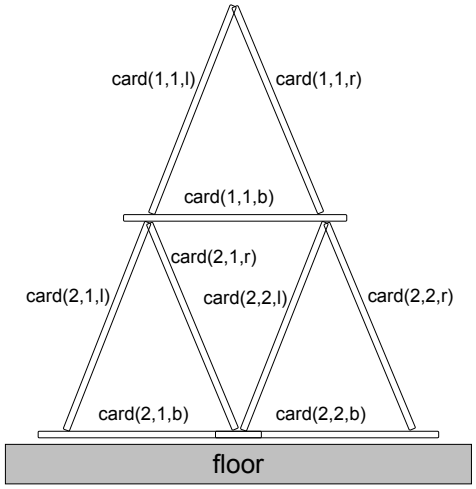
The test query `?- w(e100,f100)` results in the abducible `abd_r(a100, e100, f100)`; notice that an implementation trying out clauses in textual order will need to go through all tables many times in order to find the right place to insert a new fact. Abducibles in the store are always ground, which means that a translation is possible into PROLOG using assert-retract and hand-coded integrity constraints. Ground abducibles are also a requirement for the use of PROLOG's negation as failure above to work correctly. This program is also tested with ASYSTEM, CIFF, SMODELS and DEMOII.

The integrity constraints can be written in a more elegant way in other systems, which allows non-abducible predicates in the left hand side of integrity constraints; here we show the CIFF version.

```
[abd_p(A,B,_,_), p(A,B,_,_)] implies [false].  % A,B key
[abd_q(C,D,_,_), q(C,D,_,_)] implies [false].  % C,D key
[abd_r(A,E,_), r(A,E,_)]     implies [false].  % A,E key
```

Overall, this example is characterized by a large base of facts which is accessed very frequently (and where implementations that employ PROLOG's indexing are expected to be faster), lot of abducibles being generated, with integrity constraints based on term (non-)equality, and plenty of failed alternatives. However, there are at most two abducibles in existence at that same time. □

Example 7.2 (Sample program: build(n)) This is an extension of the planning problem of example 3.1 to build houses of cards of height n, as illustrated here for $n = 2$.



The top floor has number 1, and successively downwards and each "triangle" is numbered 1, 2, ... from left to right in each floor. Each card is represented by a term `card(`*floor*`,`*triangle*`,`x`)` where x is one of `l`(eft), `r`(ight) and `b`(ottom).

The build(n) program has O(n^2) facts describing which cards support others, and the task is to produce a plan for building the house, cf. example 3.1 above. We have extended the problem with an additional integrity constraint so that the two slanting cards touching each other in the top must be placed at successive moments. In CHR, the integrity constraints are as follows.

```
put_in_place(P0,Time0), put_in_place(P1,Time1) ==>
    supports(P0,P1), Time0 > Time1 | fail.
put_in_place(card(N,L,l),Time0), put_in_place(card(N,L,r),Time1) ==>
    1 is abs(Time1-Time0).
```

Cards are selected successively from a list which has been ordered in a non-optimal way so that a very large number of wrong choices need to be ruled out by the integrity constraints. Abducibles in the store are always ground, which means that a translation is possible into Prolog using assert-retract and hand-coded integrity constraints; this can be done as follows.

```
:- dynamic dyna_put_in_place/2.

put_in_place(C,T):- dyna_put_in_place(C,T), !.

put_in_place(C,T):-
    ( supports(C,C1), dyna_put_in_place(C1,T1), T > T1 -> fail ; true ),
    ( supports(C1,C), dyna_put_in_place(C1,T1), T1 > T -> fail ; true ),
    ( C=card(N,L,LorR), dyna_put_in_place(card(N,L,LorR1),T1),
        LorR\==b, LorR1\==b, 1 =\= abs(T-T1)
      -> fail ; true),
    ( assert((dyna_put_in_place(C,T)))
      ; retract((dyna_put_in_place(C,_))) ).
```

The CIFF integrity constraints are encoded in the following way.

```
[put_in_place(P0,Time0), put_in_place(P1,Time1),
    supports(P0,P1), Time0 #> Time1]   implies [false].
[put_in_place(card(N,L,l),Time0), put_in_place(card(N,L,r),Time1),
    (Time1-Time0)*(Time0-Time1) #\= -1]   implies [false].
```

Overall, this example is characterized by a large base of facts with a lot of abducibles being generated, with integrity constraints involving integer arithmetic, and multitudes of failed alternatives. □

Example 7.3 (Sample program: graph(n)) This program is introduced in example 3.4 above. The version for PROLOG+CHR uses SICSTUS' `clpq` library. The program can be rewritten directly into the syntax of CIFF, which handles the constraints using SICSTUS' finite domain solver. We test only for small graphs, so this example is characterized by small bases of facts, but with potentially many possibilities that need to be tried out. Integrity checking depends on an external constraint solver. □

Example 7.4 (Sample program: nqueens(n)) This is adapted from an example program for solving the classical n-queens problem which has been used in papers about the CIFF system [28] and the ASYSTEM [26] to demonstrate an effective interaction with an external constraint solver, which is SICSTUS PROLOG's finite domain solver as explained above. The CIFF version is as follows, here shown for $n = 20$; the version for ASYSTEM is almost identical in appearance.

```
abducible(q_pos(_,_)).
q_domain(R) :- R #>= 1, R #=< 20.
exists_q(R) :- q_pos(R,C),q_domain(C).
safe(R1,C1,R2,C2) :- C1#\=C2, R1+C1#\=R2+C2, C1-R1#\=C2-R2.
[q_pos(R1,C1),q_pos(R2,C2),R1#<R2] implies [safe(R1,C1,R2,C2)].
q:- exists_q(1), ..., exists_q(20).
```

This program essentially translates the problem into one set of constraints, and the finite domain constraint solver's labeling toolbox is applied in different ways by different systems for solving the problem.

In the adaptation to PROLOG+CHR, we run an abductive phase producing the constraints followed by a labeling of all variables occurring in `q_pos/2` constraints (using the `ffc` option, cf. [10]). Finally, we compare with an implementation written directly in PROLOG which produces an "optimal" set of constraints without the symmetric and redundant constraints produced by the other versions; then a labeling phase (using the `ffc` option) is initiated. We include a number of tests for different values of n in order to indicate that the time complexity is not as regular as it may appear from the referenced papers; see also [61] for an analysis of this example. □

7.3 Tests and benchmarks

All tests were made on a Macintosh Intel Core Duo, 2GHz, 2GB Ram running Mac OS X 10.4.11. PROLOG+CHR and PROLOG with assert-retract tests made with SICSTUS 4.0.1 (i386-darwin-8.9.1); PROLOG with clpfd, ASYSTEM and CIFF with SICSTUS 3.12.8 (i386-darwin-8.9.1); SMODELS is a standalone application written in C. Times are in milliseconds unless otherwise stated; a dash indicates that a test is not applicable and "∞" that the query did not terminate in observable time (1h or more). For tests involving SICSTUS PROLOG, we measure CPU time excluding garbage collecting etc., as specified by its option `statistics(runtime,_)`; see [10]. For SMODELS, the total execution is measured for both grounding and model construction.

As it appears in the table, there are many unexpected values and outliers, so all tests have been checked several times, and those involving SICSTUS PROLOG have been replicated in a Windows-based environment, showing an identical distribution of relative times used.

	PROLOG+CHR	PROLOG with assert-retract[(1)]	Simple meta-interp.[(1)]	PROLOG with clpfd	ASYSTEM	CIFF[(2)]	SMODELS
db update[(3)]	0.43	0.034	4.8	-	13	170	600
build(10)[(4)]	340	10	50	-	n/a[(5)]	530	∞[(8)]
build(15)[(4)]	3350	100	195	-	n/a[(5)]	2250	∞[(8)]
graph(3)[6]	180	-	-	-	n/a[(5)]	4020	∞[(8)]
graph(4)[(6)]	2	-	-	-	n/a[(5)]	1160	∞[(8)]
graph(5)[(6)]	14	-	-	-	n/a[(5)]	2880	∞[(8)]
nqueens(20)[(7)]	35	-	-	5	60	80	∞[(8,9)]
nqueens(32)[(7)]	45	-	-	10	180	220	∞[(8,9)]
nqueens(64)[(7)]	1300	-	-	94	990	920	∞[(8,9)]
nqueens(99)[(7)]	∞	-	-	31m	11m	12m	∞[(8,9)]
nqueens(100)[(7)]	450	-	-	108	60	2390	∞[(8,9)]
nqueens(101)[(7)]	37m	-	-	1m15s	36s	40s	∞[(8,9)]

Notes

(1) These versions can only handle ground abducibles.

(2) CIFF has two modes of treating integrity constraints, with or without grounding; the figures indicate the best setting in each case.

(3) As expected, PROLOG+CHR runs considerably faster than meta-interpretation-based systems, but the difference to the much faster PROLOG with assert-retract is striking; the cost of using CHR is a factor of 13. CIFF and SMODELS are up to 18,000 times slower than the fastest one.

(4) As opposed to db update, the simple meta-interpreter runs faster for the tested build(n) problems, which is a bit surprising. To explain this, notice that the heads of the CHR rules for integrity constraints in the PROLOG+CHR version of db update contains only a single literal. This means that CHR's search for rules to apply for an abducible (CHR constraint) amounts more or less to a call to a PROLOG predicate, whereas the build(n) problems have two head literals in their integrity rules, which means that the constraint store must be searched for the second of those, when a given call is executed. The similar search made by the simple metainterpreter is done as sequential list search which is faster for small constraint stores than CHR's more general search mechanism.

For the (again superior) assert-retract approach, the entering of an abducible is a PROLOG call, and the search for companion abducibles for checking integrity is also made as a PROLOG call, which means that PROLOG's inherent indexing is employed as far as possible.

(5) The ASYSTEM works fine for the nqueens problems and db update, but strange behaviour was observed for the build and graph problems. This may be due to the fact that ASYSTEM was developed in an earlier version of SICSTUS PROLOG that was not available for our tests.

(6) The graph(n) problem does not become more complex with larger n as there are many more paths to select from, and thus it is easier to find three paths that satisfy the conditions.

CIFF runs considerably slower that PROLOG+CHR, which may be due to the fact that different constraint solvers are used: the CLP(Q) solver applied with PROLOG+CHR has inherently incremental consistency checking, while CIFF's finite domain solver only finds inconsistencies when a labeling phase is initiated.

(7) All tests of nqueens(n), except for SMODELS, apply SICSTUS PROLOG's finite domain solver. We notice that the time does not grow monotonically with n, and there is no consistent correlation between the different implementations. The actual time depends on the labeling strategy used and whether it is applied incrementally for testing consistency, and it seems that all the strategies tested have their "black holes".

The running times indicate also that the difficulty in finding the first solution depends on n in a tricky way; for example, the case $n = 99$ is the most difficult one tested, while $n = 100$ is an especially easy case. The

PROLOG+CHR implementation is clearly more efficient for small n due to its direct execution, but the detailed control strategies for the constraint solver applied in ASYSTEM and CIFF pay back for the larger values of n.

(8) SMODELS applies a grounding preprocessing phase which enumerates all possible ground instance of predicate calls. The number of such instances grows exponentially with the number of variables in a clause. For build(10), we have integrity constraints with four variables, each of which can be shown to range over 166 different values, yielding around $8 * 10^8$ different instances which is prohibitive; a similar analysis can be made for graph(n). Furthermore, the grounding phase excludes the use of lists with standard predicates such as `member` and `append`.

(9) SMODELS could produce a solution to nqueens(8); grounding took 0.17s and model construction 0.14s, which is much slower than any of other approaches tested.

8 Concluding remarks

While CHR was originally conceived as a declarative programming language for writing constraint solvers, it has proved to be suited for what we have called hypothesis-based reasoning. We have taken a top-down approach, defining high-level patterns such as abduction and shown how their semantics can be realized by straightforward mappings into PROLOG with CHR.

This provides generally a high efficiency when compared to implementations based on meta-interpretation, especially when the problem at hand is dominated by deductive steps (such as the db update problem, example 7.1 above) in which case the major part of the work is executed directly by the PROLOG engine, employing its optimizing compiler.

The main advantages of using PROLOG and CHR in this way may be the very flexible architecture which makes it possible to 1) implement high-level patterns such as abduction in straightforward ways, and 2) "massage" such a pattern to work with different sorts of knowledge bases (such as assumptions) and integrate with existing constraint-solvers in a seamless way. On the negative side, we notice a limited support for negation when we insist on a direct execution, and our tests indicate that some of our CHR programs could run considerably faster with better indexing techniques in the underlying engine. However, any future improvement of CHR concerning its efficiency and facilities will be of benefit for hypothesis-based reasoning, whether programmed directly in PROLOG and CHR or compiled from high-level specifications into such a platform.

Acknowledgement: This work is supported by the CONTROL project, funded by Danish Natural Science Research Council.

References

[1] A. C. Kakas, R. A. Kowalski, F. Toni, Abductive logic programming, Journal of Logic and Computation 2 (1993) 719–770.

[2] A. C. Kakas, R. A. Kowalski, F. Toni, The role of abduction in logic programming, in: D. Gabbay, C. Hogger, J. Robinson (Eds.), Handbook of Artificial Intelligence and Logic Programming, Vol. 5, Clarendon Press, 1998, pp. 235–324.

[3] M. Denecker, A. C. Kakas (Eds.), Journal of Logic Programming. Special issue: abductive logic programming, Vol. 44 (1-3), 2000.

[4] M. Denecker, A. C. Kakas, Abduction in logic programming, in: A. C. Kakas, F. Sadri (Eds.), Computational Logic: Logic Programming and Beyond, Vol. 2407 of Lecture Notes in Computer Science, Springer, 2002, pp. 402–436.

[5] T. Frühwirth, Theory and practice of constraint handling rules, special issue on constraint logic programming, Journal of Logic Programming 37 (1–3) (1998) 95–138.

[6] S. Abdennadher, H. Christiansen, An experimental CLP platform for integrity constraints and abduction, in: Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series, Physica-Verlag (Springer), 2000, pp. 141–152.

[7] H. Christiansen, V. Dahl, Logic grammars for diagnosis and repair, International Journal on Artificial Intelligence Tools 12 (3) (2003) 227–248.

[8] H. Christiansen, V. Dahl, Meaning in Context, in: A. Dey, B. Kokinov, D. Leake, R. Turner (Eds.), Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05), Vol. 3554 of Lecture Notes in Artificial Intelligence, 2005, pp. 97–111.

[9] H. Christiansen, V. Dahl, HYPROLOG: A new logic programming language with assumptions and abduction, in: Gabbrielli and Gupta [62], pp. 159–173.

[10] Swedish Institute of Computer Science, SICStus Prolog Website, http://www.sics.se/isl/sicstuswww/site/index.html (checked 2007).

[11] SWI Prolog Organization, SWI Prolog Website, http://www.swi-prolog.org/ (checked 2007).

[12] H. Betz, T. W. Frühwirth, A linear-logic semantics for constraint handling rules, in: P. van Beek (Ed.), CP, Vol. 3709 of Lecture Notes in Computer Science, Springer, 2005, pp. 137–151.

[13] S. Abdennadher, H. Schütz, CHR$^{\vee}$: A flexible query language., in: T. Andreasen, H. Christiansen, H. L. Larsen (Eds.), Flexible Query Answering Systems, FQAS'98, Vol. 1495 of Lecture Notes in Computer Science, Springer, 1998, pp. 1–14.

[14] G. J. Duck, P. J. Stuckey, M. J. G. de la Banda, C. Holzbaur, The refined operational semantics of Constraint Handling Rules, in: B. Demoen, V. Lifschitz (Eds.), Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings, Vol. 3132 of Lecture Notes in Computer Science, Springer, 2004, pp. 90–104.

[15] K. L. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Plemum Press, 1978, pp. 293–322.

[16] J. W. Lloyd, Foundations of logic programming (second, extended edition), Springer series in symbolic computation, Springer-Verlag, 1987.

[17] H. Christiansen, CHRG: A grammar notation based on Constraint Handling Rules, Website with source code and examples for SICStus 3, http://www.ruc.dk/~henning/chrg/ (2002).

[18] H. Christiansen, HYPROLOG: A logic programming language with assumptions and abduction. version 0.0, Website with source code and examples for SICStus 3, www://www.ruc.dk/~henning/hyprolog/ (2005).

[19] H. Christiansen, Reasoning about passive declarations in chr, in: T. Schrijvers, T. Frühwirth (Eds.), Proceedings of CHR 2005, Second Workshop on Constraint Handling Rules, Vol. 421 of Report CW, Department of Computer Science, Katholieke Universiteit, Leuven, Belgium, 2005, pp. 93–108.

[20] H. B. Enderton, A Mathematical Introduction to Logic, Academic Press, 1972.

[21] H. Christiansen, Logic programming as a framework for knowledge representation and artificial intelligence, Teaching note for the course "Artificial Intelligence and Intelligent Systems", Roskilde University, Denmark, http://www.ruc.dk/~henning/KIIS07/CourseMaterial/CourseNote.pdf (2006).

[22] E. W. Dijkstra, Hierarchical ordering of sequential processes, Acta Informatica 1 (1971) 115–138.

[23] M. Gavanelli, E. Lamma, P. Mello, P. Torroni, An abductive framework for information exchange in multi-agent systems, in: J. Dix, J. A. Leite (Eds.), CLIMA IV, Vol. 3259 of Lecture Notes in Computer Science, Springer, 2004, pp. 34–52.

[24] H. Christiansen, CHR Grammars, Int'l Journal on Theory and Practice of Logic Programming 5 (4-5) (2005) 467–501.

[25] J. R. Hobbs, M. E. Stickel, D. E. Appelt, P. A. Martin, Interpretation as abduction., Artificial Intelligence 63 (1-2) (1993) 69–142.

[26] A. C. Kakas, B. V. Nuffelen, M. Denecker, A-system: Problem solving through abduction, in: B. Nebel (Ed.), IJCAI, Morgan Kaufmann, 2001, pp. 591–596.

[27] A. C. Kakas, A. Michael, C. Mourlas, Aclp: Abductive constraint logic programming, Journal of Logic Programming 44 (1-3) (2000) 129–177.

[28] P. Mancarella, F. Sadri, G. Terreni, F. Toni, Programming applications in CIFF, in: C. Baral, G. Brewka, J. S. Schlipf (Eds.), LPNMR, Vol. 4483 of Lecture Notes in Computer Science, Springer, 2007, pp. 284–289.

[29] C. Holzbaur, OFAI clp(q,r) Manual, Edition 1.3.3, Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna (1995).

[30] V. Dahl, P. Tarau, R. Li, Assumption grammars for processing natural language, in: Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming, MIT Press, 1997, pp. 256–270.

[31] H. Christiansen, C. T. Have, K. Tveitane, From use cases to UML class diagrams using logic grammars and constraints, in: G. Angelova, K. Bontcheva, R. Mitkov, N. Nicolov, N. Nikolov (Eds.), RANLP 2007, International Conference: Recent Advances in Natural Language Processing: Proceedings, Shoumen, Bulgaria: INCOMA Ltd, 2007, pp. 128–132.

[32] H. Christiansen, C. T. Have, K. Tveitane, Reasoning about use cases using logic grammars and constraints, in: H. Christiansen, J. Villadsen (Eds.), Proceedings of the 4th International Workshop on Constraints and Language Processing, CSLP 2007, Vol. 113 of Computer Science Research Report, Roskilde University, 2007, pp. 40–52.

[33] R. Mitkov, Anaphora Resolution, Longman (Pearson Education), 2002.

[34] L. Caroprese, I. Trubitsyna, E. Zumpano, Prioritized reasoning in logic programming, in: D. Wilson, G. Sutcliffe (Eds.), FLAIRS Conference, AAAI Press, 2007, pp. 178–179.

[35] C. Sakama, K. Inoue, Representing priorities in logic programs, in: M. J. Maher (Ed.), Logic Programing, Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming, 1996, pp. 82–96.

[36] Y. Zhang, Two results for prioritized logic programming, Theory and Practice of Logic Programming 3 (2) (2003) 223–242.

[37] L. D. Koninck, T. Schrijvers, B. Demoen, User-definable rule priorities for CHR, in: M. Leuschel, A. Podelski (Eds.), Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, ACM, 2007, pp. 25–36.

[38] H. Christiansen, On the implementation of global abduction, in: K. Inoue, K. Satoh, F. Toni (Eds.), CLIMA VII, Vol. 4371 of Lecture Notes in Computer Science, Springer, 2006, pp. 226–245.

[39] K. Satoh, "All's well that ends well" - a proposal of global abduction., in: J. P. Delgrande, T. Schaub (Eds.), 10th International Workshop on Non-Monotonic Reasoning (NMR 2004), 2004, pp. 360–367.

[40] K. Satoh, An application of global abduction to an information agent which modifies a plan upon failure - preliminary report., in: J. A. Leite, P. Torroni (Eds.), CLIMA V, Vol. 3487 of Lecture Notes in Computer Science, Springer, 2004, pp. 213–229.

[41] L. Console, D. T. Dupré, P. Torasso, On the relationship between abduction and deduction, Journal of Logic and Computation 1 (5) (1991) 661–690.

[42] T. H. Fung, R. A. Kowalski, The iff proof procedure for abductive logic programming, Journal of Logic Programmming 33 (2) (1997) 151–165.

[43] T. A. Frühwirth, Description logic and rules the CHR way, in: K. Djelloul, G. J. Duck, M. Sulzmann (Eds.), Proceedings of the 4th Workshop on Constraint Handling Rules, CHR 2007, 2007, pp. 49–61.

[44] H. Christiansen, Automated reasoning with a constraint-based metainterpreter, Journal of Logic Programmming 37 (1-3) (1998) 213–254.

[45] H. Christiansen, D. Martinenghi, DemoII System, Website with source code and examples for SICStus 3.7, http://webhotel.ruc.dk/dat/software/index.htm (1999).

[46] H. Christiansen, D. Martinenghi, Symbolic constraints for meta-logic programming, Applied Artificial Intelligence 14 (4) (2000) 345–367.

[47] V. Dahl, B. Gu, Semantic property grammars for knowledge extraction from biomedical text, in: S. Etalle, M. Truszczynski (Eds.), ICLP, Vol. 4079 of Lecture Notes in Computer Science, Springer, 2006, pp. 442–443.

[48] V. Dahl, An abductive treatment of long distance dependencies in chr, in: H. Christiansen, P. R. Skadhauge, J. Villadsen (Eds.), Constraint Solving and Language Processing, Vol. 3438 of Lecture Notes in Computer Science, Springer, 2004, pp. 17–31.

[49] V. Dahl, P. Blache, Implantation de grammaires de propriétés en CHR, in: F. Mesnard (Ed.), Programmation en logique avec contraintes, JFPLC 2004, Hermes, 2004.

[50] F. Morawietz, Chart parsing and constraint programming, in: COLING, Morgan Kaufmann, 2000, pp. 551–557.

[51] F. Morawietz, P. Blache, Parsing natural languages with CHR, Unpublished (2002).

[52] G. Penn, Applying constraint handling rules to HPSG, in: Proceedings of the First International Conference on Computational Logic (CL2000), Workshop on Rule-based Constraint Reasoning and Programming, London, UK, 2000.
URL `http://www.sfs.uni-tuebingen.de/hpsg/archive/bibliography/papers/penn_appl.ps`

[53] T. Schrijvers, The Programming Language CHR, Constraint Handling Rules, Website, http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/ (from 2005).

[54] A. C. Kakas, P. Mancarella, Database updates through abduction, in: D. McLeod, R. Sacks-Davis, H.-J. Schek (Eds.), 16th International Conference on Very Large Data Bases, VLDB, Morgan Kaufmann, 1990, pp. 650–661.

[55] A. C. Kakas, P. Mancarella, Abductive logic programming, in: Proc. of the Workshop Logic Programming and Non-Monotonic Logic, LPNMR, 1990, pp. 49–61.

[56] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The dlv system for knowledge representation and reasoning, ACM Transactions on Computational Logic 7 (3) (2006) 499–562.

[57] T. Syrjänen, I. Niemelä, The Smodels system, in: T. Eiter, W. Faber, M. Truszczynski (Eds.), Proc. of the Workshop Logic Programming and Non-Monotonic Logic, LPNMR, Vol. 2173 of Lecture Notes in Computer Science, Springer, 2001, pp. 434–438.

[58] S. Baselice, P. A. Bonatti, M. Gelfond, Towards an integration of answer set and constraint solving, in: Gabbrielli and Gupta [62], pp. 52–66.

[59] Integrating Answer Programming and Constraint Logic Programming, online proceedings, http://isaim2008.unl.edu/index.php?page=proceedings.

[60] P. Tarau, BinProlog website, http://www.binnetcorp.com/BinProlog/.

[61] A. J. Fernández, P. M. Hill, A comparative study of eight constraint programming languages over the boolean and finite domains, Constraints 5 (3) (2000) 275–301.

[62] M. Gabbrielli, G. Gupta (Eds.), Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings, Vol. 3668 of Lecture Notes in Computer Science, Springer, 2005.