On the Implementation of Global Abduction

Henning Christiansen

Roskilde University, Computer Science Dept.
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

Abstract. Global Abduction (GA) is a recently proposed logical formalism for agent oriented programming which allows an agent to collect information about the world and update this in a nonmonotonic way when changes in the world are observed. A distinct feature of Global Abduction is that in case the agent needs to give up one plan, it may start a new one, or continue a suspended plan, while its beliefs learned about the world in the failed attempts persist.
This paper describes an implementation of GA in the high-level language of Constraint Handling Rules (CHR). It appears to be a first attempt to a full implementation of GA, which also confirms CHR as a powerful meta-programming language for advanced reasoning. The construction gives rise to discussing important issues of the semantics and pragmatics of Global Abduction, leading to proposal for a specific procedural semantics and architecture that seem well-suited for real-time application.

1 Introduction

Global Abduction (GA) is an extended form of logical abduction which has been proposed recently by Ken Satoh [19, 20]. As opposed to traditional Abductive Logic Programming [16], GA features the reuse of abducibles from one branch of computation in another branch. Abducibles are here believed properties of a dynamic world, so that truth of a belief may change due to an observed change in the world. If an agent follows one plan (i.e., one branch of computation) that does not give a solution, the beliefs learned are assumed still to be valid (until contradicting evidence is learned) and should be available when another plan is tried; this other plan may be a new branch of computation or one that was previously suspended, but which can be taken up again if it appears to fit with the new state of beliefs. Interestingly, GA allows not only accumulation of knowledge but also replacing a belief by its opposite when the circumstances indicate that this is relevant.

We shall not go into a detailed explanation of GA nor argue for its advantages, but all in all, we find GA a very promising approach which deserves an implementation so that more experience can be obtained. In the present paper we describe how it can be implemented with state of the art constraint logic programming technology represented by the language of Constraint Handling Rules. However, this paper is not intended as a piece of technical documentation

Proceedings of CLIMA VII,
Seventh Workshop on Computational Logic in Multi-Agent Systems,
Future University, Hakodate, Japan, May 8-9, 2006.
K. Inoue, K. Satoh, F. Toni,
Preliminary version; revised version expected to appear in post-proceedings, *LNCS* series

only, and the implementation in an appropriate high-level language provides a setting for raising and discussing in a concrete manner, important issues of GA and its use for agent modeling.

The language of Constraint Handling Rules (CHR) [13] is an extension to Prolog designed as a declarative language for writing constraint solvers for CLP systems and is now integrated in several major Prolog versions. Previous work [1, 3, 5–8, 23] has shown that CHR is a powerful language for implementing advanced logical reasoning that goes beyond standard logic programming paradigms, and in the present paper we extend these experiences to the implementation of GA.

In the present paper we concentrate on the basic mechanisms of GA, but our work is intended to lead to an implementation of GA based on a high-level syntax and which employs the facilities of a full scale Prolog system to have GA programs communicate online with its environment. The code explained below has been developed and tested in SICStus Prolog [24], which provides the *de facto* reference implementation of CHR. Our system is still at a prototype stage with no interesting applications developed so far, but it indicates that full scale implementations are within reach with a reasonable amount of effort.

Section 2 explains the basic notions of GA, and we refer to [20] for a complete description; we define also notions of soundness and completeness for procedures and provide definitions used later for discussing flow of control and floundering. Section 3 gives a brief introduction to CHR with an emphasis on its advantages as meta-programming language for logical reasoning. In section 4 we explain how belief states can be represented and maintained, and explain the implementation of a multiple-process architecture for GA, and section 5 extends with integrity constraints. An proposal for extending GA with real world monitoring and forward reasoning is given in section 6. Section 7 uses the implementation described so far to analyze and give proposals for selection rules and flow of control; we end up advocating range-restricted GA programs with left-to-right execution as a suitable choice that respects the pragmatic considerations underlying GA. We add to section 7 also an explanation of how a cut operator indicated by [19, 20] can be added to GA and implemented in the left-to-right-setting. A final section provides a summary and future direction.

2 Global Abduction

Disjoint sets of *belief predicates*, *ordinary predicates*, and *equality* and *nonequality predicates* $=$ and $\neq$ are assumed, the latter two having their standard syntactic meaning; we allow positive and negative literals over belief predicates but not over the other categories (for which "literal", thus, indicates "atom" only). We define also *annotated literals* of the form `announce(`B`)` and `hear(`B`)`, where B is a belief literal. A notation of prefix plus and minus is applied for positive and negative belief literals. *Program clauses* are defined as usual, expected to have an ordinary atom as its head and a conjunction of zero or more literals as body. An *integrity constraint* is a clause whose head is `false` (expressing falsity), and with only belief literals in the body. A *global abductive framework*

(GAF) $\langle \mathcal{B}, \mathcal{P}, \mathcal{IC} \rangle$ consists of a sets of belief predicates $\mathcal{B}$, program clauses $\mathcal{P}$, and integrity constraints $\mathcal{IC}$. A *belief set* is a set of ground literals over belief predicates which contains no pair of contradictory literals.

Intuitively, an announcing literal `announce(`L`)` for ground belief L, means to assert L in a current belief set so that a hearing literal `hear(`L`)` becomes satisfied in that belief set. The truth value of a belief literal not in the given belief set is recognized as unknown, and similarly for corresponding annotated literals. We introduce the semantics informally by an example; it anticipates left-to-right execution but it should be stressed that this is not assumed for the declarative semantics of [19].

Example 1. The following program is supposed to help an agent to decide whether or not to cross the street; `sS` stands for "stop signal" and `t` for "traffic".

```
decide(walk):- hear(-sS), hear(-t).
decide(wait):- hear(+sS).
check:- announce(-sS), announce(+t).
check:- announce(+sS).
```

The initial query is `check, decide(X)` which corresponds to a process which may split into two, one for each of the two `check` clauses.
(1) `announce(-sS), announce(+t), decide(X)`
(2) `announce(+sS), decide(X)`
Each process has its own set of *belief assumptions* (BA) which must be consistent with the *current* global *belief state* (CBS) in order to classify as *active*. The BA of a process consists of those beliefs it has applied in getting to its present state; the CBS consists of `announce`d beliefs, maintained in a nonmonotonic way and independently of which processes made the `announce`ments.

Suppose (1) executes the two `announce`ments leading to process
($1'$) `decide(X)` with $BA = \{$`-sS`, `+t`$\}$ and
$CBS = \{$`-sS`, `+t`$\}$. Process ($1'$) splits into ($1'.1$) and ($1'.2$).
($1'.1$) `hear(-sS), hear(-t)` with binding `X=walk` and $BA = \{$`-sS`, `+t`$\}$.
($1'.2$) `hear(+sS)` with binding `X=wait` and $BA = \{$`-sS`, `+t`$\}$.
Process ($1'.1$) continues with `hear(-sS)` but gets stuck on `hear(-t)` which does not fit with CBS; ($1'.2$) gets stuck in a similar way. Thus branches ($1 \cdots$) failed to provide a solution, but the CBS persists. Now process (2) may be tried, doing `announce(+sS)`, which results in a revision of CBS:
$CBS = (\{$`-sS`, `+t`$\}$ revised with `+sS`$) = \{$`+sS`, `+t`$\}$.

Left of (2) is `decide(X)`, which via 2nd clause executes successfully a `hear` and terminates with the binding `X=wait` and the final belief set $\{$`+sS`, `+t`$\}$.

A declarative semantics for GA is given in [19], based on a three-valued minimal model approach [12, 18]. Truth value of a statement ϕ in a GAF $\langle \mathcal{B}, \mathcal{P}, \mathcal{IC} \rangle$ is expressed relative to a belief set Bs, written$\langle \mathcal{B}, \mathcal{P}, \mathcal{IC} \rangle \models_{Bs} \phi$.

By a *proof procedure*, which may exist as an implemented program, we indicate a device which given a GAF and an initial goal G produces zero or more *answers*, each of which is a pair $\langle Bs, \sigma \rangle$ of belief set Bs and substitution σ to the variables of G; a *goal* is a set (or sequence, depending on context) of literals.

Definition 1. *Let a GAF* $GA = \langle \mathcal{B}, \mathcal{P}, \mathcal{IC} \rangle$ *and an initial goal* G *be given, and assume a fixed proof procedure.*

- *A* correct answer *for* G *in* GA *is one* $\langle Bs, \sigma \rangle$ *so that* $\langle \mathcal{B}, \mathcal{P}, \mathcal{IC} \rangle \models_{Bs} G\sigma\rho$ *for any grounding substitution* ρ*.*
- *A* computed answer *for* G *in* GA *is one returned by the proof procedure.*
- *A proof procedure is* sound *whenever any computed answer is also a correct answer.*
- *A proof procedure is* complete *whenever, for any correct answer* $\langle Bs, \sigma \rangle$*, there exists a computed answer* $\langle Bs', \sigma' \rangle$ *so that* $Bs' \subseteq Bs$ *and* $\sigma = \sigma'\rho$ *for some substitution* ρ*.*

GA has some inherent problems concerning the relationship between correct answers and computed answers not recognized in [19]. We illustrate this with the following queries (variables existentially quantified); the first one is not problematic but the remaining ones are:

1. `announce(+b(1)), hear(+b(1))`
2. `announce(+b(X)), hear(+b(1))`
3. `announce(+b(1)), hear(+b(X))`

No. 1 is clearly satisfied with the belief set {`+b(1)`} and we would expect any capable implementation to recognize this. No's 2–3 are also satisfiable with {`+b(1)`} by substitution `X` $\mapsto$ `1`. The procedural semantics of [19] does not recognize the solutions for 2–3 but ends in floundering states.

We may, however, ask whether it is reasonable to expect them to be found. As presented intuitively and illustrated by the examples of [19, 20], `announce` is an event that goes "before" `hear`, and `announce` indicates changes in the agent's belief about the world. So producing `X` $\mapsto$ `1` from case 2 is a bit absurd as this corresponds to an agent for whom an act of Providence modifies the world to fit what it wants to hear. On the other hand, we may suggest an extension that produces the solution for case 3, which seems intuitively correct when considering `hear(+b(X))` as a query for information about the current status for belief predicate `b/1`; we return to this topic in section 7.

We adapt the notion of range-restriction familiar from deductive databases in order to impose restrictions on which GAFs we prefer. Range restriction in a database implies that any query or relation has a well-defined and finite extension derivable from base relations; for a knowledge based system such as GA, these database heuristics appear quite reasonable. We define a left-to-right version of this notion which ensures that every variable achieves a definite value under left-to-right, thus reducing the risk of floundering under left-to-right execution, and we end up later concluding that left-to-right fits well with a pragmatic understanding of GA.

Definition 2. *A clause, an integrity constraint, or a query (considered as a clause with empty head) is* range-restricted *(rr) whenever any variable in it appears (as well) in an ordinary literal in its body; it is* `hear` range-restricted *(*`hear`*-rr) whenever any variable in it appears (as well) in an ordinary or* `hear`

literal in its body; it is left-to-right range-restricted *(lr-rr) whenever any variable in it has a leftmost occurrence within the body in an ordinary literal; it is* left-to-right `hear` range-restricted *(lr-*`hear`*-rr) whenever any variable in it has a leftmost occurrence within the body in an ordinary or* `hear` *literal.*

A GAF has one of these properties if all its clauses, integrity constraints, and permitted queries have the property.

Example 2. The following clause is *lr-rr*.

```
p(X,Y):- q(X), r(Y), announce(+Y).
```

The next one is *lr-*`hear`*-rr* but not *lr-rr*.

```
p(X,Y):- q(X), hear(-Y).
```

The clauses `p(X,Y):- q(X)` and `p:- announce(+X)` have none of these properties.

The primary source on GA, [19], proposes a proof procedure which is not reproduced here. Our implementation is explained below in a self-contained way but for the interested reader, we refer to the steps of that proof procedure writing, e.g., PP3.5 for its case 3.5. We refer to the following central notions: an *execution state* consists of a set of *processes* and a *current belief set*. A *process* is a triplet consisting of a *current goal* (a query), a set of belief literals called its *belief assumptions* that has been used by this process so far, and an answer substitution. A process is *active* if its belief assumptions are true in the current belief set, *suspended* otherwise.

3 Constraint Handling Rules as a Metalanguage for Logic-based Systems

Constraints of CHR are first-order atoms whose predicates are designated constraint predicates, and a constraint store is a set of such constraints. CHR is integrated with Prolog and when a constraint is called, the constraint solver defined by the programmer takes over control. A constraint solver is defined in terms of rules which can be of the following two kinds.

$$\text{Simplification rules: } c_1, \ldots c_n \texttt{ <=> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$
$$\text{Propagation rules: } c_1, \ldots c_n \texttt{ ==> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$

The c's are atoms that represent constraints, possibly with variables, and a simplification rule works by replacing in the constraint store, a possible set of constraints that matches the pattern given by the *head* $c_1, \ldots c_n$ by those corresponding constraints given by the *body* $c_{n+1}, \ldots, c_m$, however only if the condition given by *Guard* holds. A propagation rule executes in a similar way but without removing the head constraints from the store. The declarative semantics is hinted by the applied arrow symbols (bi-implication, resp., implication formulas, with variables assumed to be universally quantified) and it can be shown that the indicated procedural semantics agrees with this. There is also a mixture

of the two rules, called *simpagation* $cs_1 \backslash cs_2$ `<=>` $\cdots$ in which cs_1 is kept in the store and cs_2 removed when such a rule applies. Guards should be tests (with no unifications to head variables and no calls to constraints), and rule bodies can, in fact, be any executable Prolog term and is executed as such. Procedurally, CHR is a committed choice language which means that a failure (in Prolog's sense) within the body of a CHR rule means failure of the goal that triggered that rule, but backtracking remains in Prolog constructs as usual. Notice that CHR uses a one-way matching and not unification when a rule applies for a set of constraints. When a constraint is called, rules are tested for applicability in textual order, and when a rule is chosen, its body finishes before the possibly next rule for that constraint is allowed to start.

CHR is more powerful than indicated by its declarative semantics, as the constraint store can be used as a global resource that can be modified in non-monotonic ways. A simplification rule may, for example, remove a piece of information and replace it with complementary information. In this way we may use the constraint store for, say, a pool of processes and for belief sets. Furthermore, CHR programmers tend to employ their procedural knowledge about the sequential order in which rules are tried, as we do in the following; this has inspired to the formulation of a formal semantics, the so-called refined operational semantics [11], that takes all these aspects into account. There is an interesting analogy between the view of the constraint store as a global resource and a recently proposed semantics for CHR based on linear logic [4]; this relationship, however, has not studied further.

As indicated by [2], CHR makes it possible to combine different control strategies in an easy way, and for GA we need to make use of process delays and breadth-first search. For the latter, we need to be careful in catching Prolog-failures in one branch so they do not destroy other branches; in other words, a process with a failure should be eliminated, i.e., avoid calling a continuation in case of a failure. This can be done in the following way using Prolog's conditional construct (for background of such transformation of CHR rules, see [14]).

Head `==>` *Guard* `|` `(`*SuccessOrFailure* `->` *Continuation* `;` `true).`

In most well-behaved CHR programs, all references to the constraint store are normally made indirectly by the matching in head of rules. However, in some cases (as in our code, below), it may be useful to test explicitly whether a certain constraint is in the store; CHR has a primitive called `find_constraint` for this purpose.

For the meta-programming task at hand, we need to do renaming of variables which is done using a predicate `renameVars/2` which produces a copy of its first argument with all variables consistently replaced by other variables. It is implemented by the one-liner `renameVars(A,B):- asserta(dummy(A)), retract(dummy(B)).`

Finally, we remind the reader that there is a the risk of confusion due to terminological overlap: a *constraint* of CHR is a single atom, while an *integrity constraint* in general is a complex condition which is some cases can be identified with a *rule* of CHR.

4 Belief States and Processes

GA's belief literals are represented by means of two constraints, indicated by symbols plus and minus (which can be written as prefix operators) for positive and negative literals. A current belief state, then, is represented as the set of such constraints in the store. According to PP3.5, adding a belief literal to the state should remove any possible opposite literal; the complete code necessary for managing the belief state, including rules that remove duplicates, is as follows.

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

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

The `pragma passive` syntax indicates that only a call to a predicate matching an occurrence which is *not* marked as passive is able to trigger that rule; see [24] for a detailed explanation. It is applied here to keep only the most recently announced belief in store and to remove a possible earlier contradictory belief. These rules must precede any others so that process rules are never triggered in a state with contradictory beliefs.

Processes are represented in the store by declaring a constraint `process/3` whose arguments are as follows.

`process(`*Goals*`,` *BeliefAssumptions*`,` *Bindings*`)`

The first argument is a list containing the remaining subgoals to be resolved by that process; 2nd arg. is the process' assumptions given as a list of beliefs; 3rd arg. the bindings of variables of the initial query.

We assume a predicate `inCurrent/1` which given the assumptions of a process tests (using `find_constraint`) whether they hold in the current belief state; this implements GA's test for active processes. This predicate is used as a guard in every rule that describes a computational step of a process. We explain at the end of the section how the binding argument is handled; it can be ignored for the understanding of the overall principles of the procedure. The following rules capture PP1, PP3 except PP3.1; the handling of integrity constraints in PP3.5 is postponed until section 5. The `add` predicate adds an element to a list but avoiding duplicates. The rules below select always the left-most literal for execution, but other alternatives are discussed later.

```
process([],As,Bind) <=> inCurrent(As) | write(Bind), success.

process([(X=Y)|Gs],As,Bind) <=> inCurrent(As) |
    (X=Y -> process(Gs,As,Bind) ; true).

process([(X\=Y)|Gs],As,Bind) <=> ?=(X,Y), inCurrent(As) |
    X=Y -> true ; process(Gs,As,Bind)).

process([hear(B)|Gs],As,Bind) <=> ground(B) |
    add(B,As,NewAs), process(Gs,NewAs, Bind).

process([announce(B)|Gs],As,Bind) <=> ground(B), inCurrent(As) |
    B, add(B,As,NewAs), process(Gs,NewAs,Bind).
```

The `success` constraint removes (by rules not shown) all other `process`es from the system so that they cannot continue. Thus, after printing the bindings, the final constraint store returned by CHR will consist of exactly the final belief set plus `success`. Failure of the computation in the sense of GA is signified by a final state in which no `success` is present; this principle implements PP3.2; any other final state (with suspended `process` constraints) is "floundering".

The rules for equality and nonequality eliminate processes with Prolog-failure using the technique described in section 3 above. For nonequality, a peculiar SICStus Prolog test `?=(X,Y)` is used [24]. It is satisfied whenever `X` and `Y` are either identical or sufficiently instantiated as to tell them different; this makes our procedure go a bit further that Satoh's proof procedure which has to wait until both arguments become ground.

Finally, we implement `hear(B)` by the little trick of adding `B` to the process' assumptions. In this way, the process needs to wait until consistency of assumptions and `B` becomes true, and this provides the behaviour of PP3.4. The rules for `hear` and `announce` can easily be extended with a check so that the process vanishes in case the belief literal `B` is incompatible with the assumptions `As`.

There is one omission in the procedure presented so far that concerns restarting of once suspended processes. A process is delayed in case the `inCurrent` test tells it passive, but it may happen later that another process changes the belief state in a way that makes the `inCurrent` test succeed.

The basic mechanisms of CHR try a rule r for applicability when either a constraint (which has a matcher in the head of r) is called or when a variable in such a stored constraint is unified. However, CHR does not test for cases where the outcome of a guard changes from false to true in the absence of the other kinds of "triggering events", as is the case with our use of `inCurrent`.[1] We implement a mechanism for this delay-and-retry phenomenon by the following rules.

```
process(Gs,As,Bind) <=> \+ inCurrent(As) |
    delayedProcess(Gs,As,Bind).

+_ \ delayedProcess(Gs,As,Bind)#Delay <=> inCurrent(As) |
    process(Gs,As,Bind) pragma passive(Delay).

-_ \ delayedProcess(Gs,As,Bind)#Delay <=> inCurrent(As) |
    process(Gs,As,Bind) pragma passive(Delay).
```

For simplicity of code, these rules perform a check for possible restart whenever the belief state changes. This can be optimized by introducing instead a specialized constraint for restart which is called whenever some process goes passive or vanishes and which locates one other delayed process for restart by using a simplification rule.

[1] Such guards are not considered not good style of CHR programming, but are difficult to avoid in the present case. In an attempt to do this by a matching rule head, we would need one specific rule for each possible set of belief assumptions with those assumptions in the head, which of course vary dynamically; hence this approach is impossible.

In order to handle the goals of ordinary predicates we need to start a new process for each clause whose head unifies with the goal in a breadth-first way. This is done by a compilation of clauses into CHR rules described as follows. Consider a predicate p/k, defined by clauses p($\overline{t_1}$):-B_1, ..., p($\overline{t_n}$):-B_n; the expressions $\overline{t_i}$ indicate sequences of k terms, B_i arbitrary bodies, and $\overline{X}$, $\overline{X}_r$ below sequences of k distinct Prolog variables. The ith clause, $i = 1, \cdots n-1$ is compiled into the following rule.

```
process([p(X̄)|Gs],As,Bind) ==>
    renameVars(f(X̄,Gs,Bind),f(X̄r,GsR,BindR)),
    append([X̄r=t̄i|Bi],GsR,NewGs),
    (inCurrent(As) -> process(NewGs,As,BindR)
                    ; delayedProcess(NewGs,As,BindR)).
```

Clause n is compiled in a similar way, except that the rule is made into a simplification rule, i.e., using `<=>` instead of `==>`. The strategy applied is, thus, that n fresh copies replace the original call p($\cdots$), so that the unifications for the different clauses can be done correctly in simultaniety.[2]

An initial query such as `p(X,Y), q(Y,Z)` is given to the system as a `process` constraints with the following arguments.

```
process([p(X,Y),q(Y,Z)],[],['X'=X,'Y'=Y,'Z'=Z])
```

The first argument is the list containing the query, the 2nd is the (still) empty list of belief assumptions (but default beliefs can be added here), and the 3rd one represents the bindings. Each "equation" represents an association between a variable (say `X` at the rhs) and its name (`'X'` at the lhs, which is a quoted constant that Prolog prints without quotes). Along each branch of computation, the variable will be affected by all unifications and renamings, but its name is not. Thus in the final state, it represents perfectly the binding made to the variable `X` appearing in the initial query.

5 Integrity Constraints in GA

In [20], integrity constraints are not used and for the examples in that paper, we can do with the implemented system described so far. The integrity constraints of [19] can be handled in a way similar to how we handled consistency of belief sets above. While consistency means that *all* beliefs in a process' assumptions are in the current belief set, a given integrity constraint is satisfied if *not all* beliefs of a certain set are in the current belief set. As for consistency, a process should delay if it violates an integrity constraint and be tested again when the current belief set changes.

[2] The test for `inCurrent(As)` made inside the body (instead of as guard) takes care of those cases where, say, the execution if B_i has changed the belief set so that the next `inCurrent(As)` for B_{i+1} fails.

Integrity constraints should be tested in the `process` rule for `announce(B)` (cf. PP3.5), and this can be done by adding to its guard an extra test `icHolds(B)`. It succeeds when, for each ground instance of an integrity constraint containing the actual `B`, the other belief literals in it are not in the current belief set. This can be implemented in a way similar to `inCurrent`, and a suitable delay-and-retry mechanism for `announce` `process`es can be implemented using the same principles as we used above for the general `inCurrent` test. The extensions to the code are straightforward and omitted.

Readers familiar with earlier work on abductive logic programming (ALP) in CHR (e.g., [1, 8]) may wonder why we did not write integrity constraints directly as rules of CHR, e.g., write "`false:- a,b`" as a propagation rule "`+a,+b==>false`" which, then, would fail when, say, `+a` is added to a state including `+b`. It seems possible to adapt the code shown above this approach as well, but a mechanism to wake processes that were blocked by failing integrity constraints may become difficult to implement.

Writing integrity constraints as CHR rules works perfectly for ALP in a combination of Prolog and CHR since in that paradigm, more and more abducibles are collected in a monotonic fashion; the indicated failures caused by integrity constraints discard effectively the current branch, and Prolog backtracks neatly to the next one. In GA, on the other hand, control may jump back and forth between different branches and the belief set may be updated in nonmonotonic ways.

6 Proposal for an Extension of AG with Monitor Processes and Forward Reasoning

By a monitor process, we indicate a process which supplies real world data into the global belief set. In this way the agent may adjust its behaviour according to events in the real world. We suggest to have separate mechanisms for defining such processes for the following reasons.

- Monitors should work independently of other processes and not wait until other processes decide to give up the control. This may be implemented by a sort of interrupt technology or, what is easily incorporated into the present CHR implementation, called with regular intervals from the rules that comprise the interpreter.
- Monitor processes should be released from the basic GA restriction of keeping their own belief assumptions consistent with the current (global) belief set. If a normal GA process executes `announce(+a)` followed by `announce(-a)` it blocks forever as its belief assumptions contains {`+a`,`-a`} which is inconsistent with any belief state. This is obviously problematic for a process monitoring real world event.

Current Prolog systems with CHR, such as SICStus [24], provide libraries which can communicate with external state-of-affairs, so we may suggest to add a special syntax as follows.

`monitor` *code*`.`

The code can inspect different external sources and perform suitable `announce`-ments and in this way affect the program execution to change its processes accordingly (the interpreter takes care to call all monitors now and then).

Furthermore, it seems useful to include forward reasoning at the level of the global belief set. This is done simply by writing CHR rules about beliefs, e.g.:

```
+robber, -police ==> +emergency.
```

If the combination of beliefs recorded in the head occurs in the current belief set, those in the body are added immediately, and this may affect the GA program, e.g., to switch process.

In the detailed design of a mechanism for forward reasoning, it should be considered whether the programmer is responsible for supplying additional rules to remove conclusions in case their premises disappear, or such additional rules should be produced by an automatic analysis.

7 Details of the Procedural Semantics

The strategies for selection of literal and process for the continuation are important choices from logical as well as pragmatic perspectives.

The procedure outlined above considers only the leftmost literal of a process for possible execution, and when an ordinary literal is expanded by means of a clause, its body is appended to the left, which yields a Prolog-like depth-first, left-to-right execution within each branch of computation.

This may obviously cause deadlocks and loss of completeness as illustrated by

```
process([hear(a),announce(a)], ...).
```

If desired, the selection of literal can be modified as to search for the leftmost one for which an execution step can apply.

We take the liberty to assume that our implementation is *sound*, independently of selection strategies. We base this claim on the fact that the implementation is modeled over the abstract proof procedure of [19] for which a soundness result is given; the referenced paper does not give completeness results. In fact, it is not complete as we indicated by example queries in section 2.

Our own procedure as well as that of [19] can be extended further for *lr*-`hear`-*rr* programs by adapting the step for `hear` so that in case of a nonground argument, it tries to unify the argument with the different current beliefs in separate processes. Another source for lack of completeness in our procedure is that it does not try out alternative interleavings of different processes. A jump from one process to another takes place only in case the leftmost literal L is blocked, either because the process is suspended as a whole or a specific condition for L is unsatisfied. A detailed analysis can show that when this happens, control will go to one of the delayed processes if any, otherwise it will move upward in

the execution tree, searching for the nearest point where an ordinary atom can try an alternative program clause.

It may be possible to obtain a complete procedure, i.e., one that eventually produces (answers that subsume) all correct answers by trying different interleavings on backtracking, or by an approach that keeps track of alternative belief states in parallel. Such implementations, however, are likely not of any practical value due to lack of efficiency. In addition, a backtracking implementation may be problematic for a program with real world monitoring.

However, it is interesting to notice in [20], that the inventor of GA restricts to a version of it that employs a Prolog-like execution strategy and even suggests the addition of a procedural device analogous to Prolog's cut. There may be good reasons for this.

First of all, GA is really about resources, `announce` *creates* a new resource, and `hear` *applies* it, and it is possible to *consume* a resource `r` by the code `hear(+r), announce(-r)`. The papers about GA [19, 20] apply usages that indicate an implicit notion of time (which is reflected in our paper as well), so it is reasonably to assume that a GA programmer has an understanding of a sequential execution. I.e., the bodies in a clause are executed from left-to-right, possible halting or interleaved with other branches, but never right-to-left or any other order.

Under these considerations, it is reasonable to restrict programs to be *lr*-`hear`-*rr*, which provides a good intuitive reading of programs with beliefs states as well as the implicit time moving from left to right.

Thinking of GAFs as a practical programming language for developing complex but fairly efficient agents, we may suggest to fix a deterministic execution strategy which makes it possible for the programmer to foresee – and optimize – the procedural behaviour of the program. This is the way things are done, and programmers tend to think, in logic program languages such as Prolog and CHR (as we have demonstrated fully above!). The execution strategy implemented in our system seems to be a good candidate for a fixed strategy, and this may conveniently be combined with a check that rejects GAFs that are not *lr*-`hear`-*rr*.

A Note on the Implementation of Cut

Cut in a breadth-first context works differently from cut in Prolog and it is not obvious how it should work. We show an example; assume a predicate `p` is defined by a set of clauses as follows.

```
p(···):- B1, !, B2, !, B3.
p(···):- B4, !, B5.
p(···):- B6.
```

In case, say, `p(a)` can unify with the head of these clauses, a process for each clause may become active, and when one of them reaches its cut, the other two should be eliminated. Assumes, as an example, that this happens in the first

clause. But at this point in time, the processes for all three clauses may have multiplied into several ones each. A voting among logic programmers will likely indicate a majority for the proposal that all process arising from the two last clauses should be eliminated. It is less clear for the different processes that arise during the processing of B_1, all having "the same" cut in their query arguments that represent their continuations.

We will take the solution that only the particular subprocess that gets to the cut first should survive. To obtain this, we associate a unique key with the particular call to `p`, which is attached to each of the three derived processes so that any process in the state keeps a list of keys referring to split-points involving cuts that may endanger it.

The split-point's key is also attached to the visible cut operators so when one of those is executed, it knows which processes to eliminate. The winning process, i.e., one that reaches a cut before being eliminated, can be handled as follows.

```
process([cut(N)|Gs],As,Bind,Cuts) <=> inCurrent(As) |
    eliminate(N), process(Gs,As,Bind,Cuts).
```

Elimination of the relevant processes can be done by the following CHR rules.

```
eliminate(N) \ process(_,_,_,Cuts)#X <=> member(N,Cuts) | true
     pragma passive(X).
eliminate(_) <=> true.
```

(A rule similar to the first one should also be added for `delayedProcesses`.) To see that it works, notice that the winning process is outside the pool when `eliminate` makes its harvest. The `eliminate` constraint triggers the simpagation rule as many times as possible to eliminate `process`es referring to the given cut, and finally it eliminates itself by the last rule; after that, the winner can safely enter the pool again. An encoding of cut by means of `announce` and an integrity constraint is exemplified in [19] but it is difficult to compare this idea with our general proposal.

8 Comparison with other Other Paradigms

It is interesting to compare with Assumptive Logic Programming [25, 10] which can be understood as Prolog extended with operators to manage global resources (called assumptions), which include counterparts to `announce` and `hear`. The specification of that language explicitly says that the operations affect the state given to the continuation, defined in the standard Prolog way. It is possible to announce linear hypothesis, meaning that they can used once (i.e., consumed), or intuitionistic ones which can be applied infinitely many times. It is also possible to refer to "timeless" assumptions which can be requested before they are announced; this is useful because assumptions and requests are matched by unification, and it is shown to have relevant applications for linguistic phenomena. Assumptive Logic Programming does not include the possibility of jumping from branch to branch and reusing beliefs, but may be interesting to propose

an extension of GA with the full repertoire of Assumptive Logic Programming's operators which provides a high flexibility for working with beliefs.

The reader may wish to compare the architecture described in the present paper with a CHR based implementation of Assumptive Logic Programming with integrity constraints and combined with traditional Abductive Logic Programming, as done in the the HYPROLOG system [7, 8].

It may, in fact, be questioned whether a more appropriate name for GA would be Global Assumptive Programming, as GA as well as Assumptive Logic Programming share the property that new knowledge is explicitly announced with specific operators and explicitly consulted with others. In traditional Abductive Logic Programming [16] (ALP), on the other hand, there is no explicit creation of knowledge: when the program refers to a particular piece of information, for the first or the nth time, the system provides an illusion that it was there already from the beginning.

Furthermore, it seems fairly straightforward and relevant to extend GA to describe systems of multiple cooperating agents. Each agent has its own program clauses and can execute them as described, including jumping from branch to branch, and the current belief state could be shared between all such agents. Alternatively, distinctions can be made between an agent's private beliefs and common beliefs. Implementationwise, each agent may run on its own physical processor, or parallelism can be simulated. Such a system seems to be useful for coordinating agents which access different external sources simultaneously in order to solve a given task.

A subset of GA can be executed within the speculative computation framework of [21, 22] which is reported to be implemented in Prolog. The mentioned frameworks as well the agent-oriented logic programming languages of [9, 17], just to mentioned a few, may also seem interesting to approach with implementations in CHR.

A proof method called Dynamic SLDNF [15] has been applied for implementing an agent-oriented architecture which allows arbitrary clauses to be removed and added; a representation of the proof tree is maintained in an incremental way when the program changes, and it is not obvious that CHR should provide special advantages here.

As we have indicated, CHR is a powerful meta-programming language for advanced reasoning demonstrated mostly by our own work [1, 5–8]; we may also refer to [3, 23]. The present paper shows that a rule-based approach to constraint programming fits well also for implementation of process-oriented languages.

9 Conclusion

We have suggested an implementation of Satoh's Global Abductive Frameworks using the high-level programming language of Constraint Handling Rules. We have described the basic principles applied in a prototype version and shown how the different aspects of the formalism can be implemented, including cut. We have learned several important things from this exercise:

– We have been able to analyze and compare different execution strategies and advocate one which we claim fits with realistic implementations as well as applications in real-time environments.
– The implementation principles are described in so much detail as to indicate that a full instrumented version with a high-level syntax[3] is within reach with a reasonable amount of effort.
– We have compared with other systems, and we have provided a catalogue of interesting and implementable extensions to Global Abduction.

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

References

1. Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pages 141–152. Physica-Verlag (Springer), 2000.
2. Slim Abdennadher and Heribert Schütz. Chrv: A flexible query language. In Troels Andreasen, Henning Christiansen, and Henrik Legind Larsen, editors, *FQAS*, volume 1495 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1998.
3. Marco Alberti, Federico Chesani, Marco Gavanelli, and Evelina Lamma. The chr-based implementation of a system for generation and confirmation of hypotheses. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 111–122. Universität Ulm, Germany, 2005.
4. Hariolf Betz and Thom W. Frühwirth. A linear-logic semantics for constraint handling rules. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2005.
5. Henning Christiansen. Abductive language interpretation as bottom-up deduction. In Shuly Wintner, editor, *Natural Language Understanding and Logic Programming*, volume 92 of *Datalogiske Skrifter*, pages 33–47, Roskilde, Denmark, July 28 2002.
6. Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 5(4-5):467–501, 2005.
7. Henning Christiansen and Veronica Dahl. Assumptions and abduction in Prolog. In Elvira Albert, Michael Hanus, Petra Hofstedt, and Peter Van Roy, editors, *3rd International Workshop on Multiparadigm Constraint Programming Languages, MultiCPL'04; At the 20th International Conference on Logic Programming, ICLP'04 Saint-Malo, France, 6-10 September, 2004*, pages 87–101, 2004.
8. Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2005.

[3] Our experiences with other systems implemented in a combination of CHR and Prolog [6, 8] demonstrate that high-level syntax is fairly straightforward to implement as soon as the underlying mechanics is in place, using operator declarations and the `term_expansion` primitive; see [24].

9. Stefania Costantini and Arianna Tocchio. A logic programming language for multi-agent systems. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *JELIA*, volume 2424 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2002.
10. Verónica Dahl, Paul Tarau, and Renwei Li. Assumption grammars for processing natural language. In *ICLP*, pages 256–270, 1997.
11. Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.
12. Melvin Fitting. A deterministic prolog fixpoint semantics. *J. Log. Program.*, 2(2):111–118, 1985.
13. Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
14. Thom W. Frühwirth and Christian Holzbaur. Source-to-source transformation for a class of expressive rules. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE*, pages 386–397, 2003.
15. Hisashi Hayashi. *Computing with Changing Logic programs.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 2001.
16. A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming,* vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pages 235–324, 1998.
17. João Leite and Luís Soares. Enhancing a multi-agent system with evolving logic programs. In Katsumi Inoue, Ken Satoh, and Francesca Toni, editors, *Seventh Workshop on Computational Logic in Multi-Agent Systems (CLIMA-VII)*, 2006. To appear.
18. Teodor C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.*, 13(4):445–463, 1990.
19. Ken Satoh. "All's well that ends well" - a proposal of global abduction. In James P. Delgrande and Torsten Schaub, editors, *NMR*, pages 360–367, 2004.
20. Ken Satoh. An application of global abduction to an information agent which modifies a plan upon failure - preliminary report. In João Alexandre Leite and Paolo Torroni, editors, *CLIMA V*, volume 3487 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2004.
21. Ken Satoh, Katsumi Inoue, Koji Iwanuma, and Chiaki Sakama. Speculative computation by abduction under incomplete communication environments. In *ICMAS*, pages 263–270. IEEE Computer Society, 2000.
22. Ken Satoh and Keiji Yamamoto. Speculative computation with multi-agent belief revision. In *AAMAS*, pages 897–904. ACM, 2002.
23. Christian Seitz, Bernhard Bauer, and Michael Berger. Multi agent systems using Constraint Handling Rules for problem solving. In Hamid R. Arabnia and Youngsong Mun, editors, *IC-AI*, pages 295–301. CSREA Press, 2002.
24. Swedish Institute of Computer Science. SICStus Prolog user's manual, Version 3.12. Most recent version available at `http://www.sics.se/isl`, 2006.
25. Paul Tarau, Verónica Dahl, and Andrew Fall. Backtrackable state with linear assumptions, continuations and hidden accumulator grammars. In John W. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium*, page 642. MIT Press, 1995.