Search Strategies in CHR(Prolog)

Leslie De Koninck*, Tom Schrijvers**, Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
{leslie,toms,bmd}@cs.kuleuven.be

Abstract We extend the refined operational semantics of the Constraint Handling Rules language to support the implementation of different search strategies. Such search strategies are necessary to build efficient Constraint Logic Programming systems. This semantics is then further refined so that it is more suitable as a basis for a trailing based implementation. We propose a source to source transformation to implement breadth first search in CHR(Prolog): CHR with Prolog as the host language. Breadth first is chosen because it exhibits the main difficulties in the implementation of search strategies, while being easy to understand. We evaluate our implementation and give directions for future work.

1 Introduction

"Algorithm = Logic + Control" is a famous quote by Robert Kowalski, implying a separation between the declarative meaning of a program and its operational behavior. The latter consists of choosing the order in which different conjunctives and disjunctives are processed. In the context of Constraint Logic Programming, the order of the conjunctives is handled by scheduling and the order of disjunctives by search. In this paper we focus on the latter.

Search strategy is considered to be a crucial component in making CLP systems efficient. The standard left-to-right depth first search which is often given to a CLP system by its underlying Prolog implementation, does not always lead to the best results. In this paper, we investigate how search strategies can be implemented in the Constraint Handling Rules (CHR) language.

CHR [8] is a high-level rule-based language, built on top of a host language like Prolog [15], Java [2], Haskell or Curry [9], and designed for a more easy implementation of Constraint Programming facilities. CHR can be extended to $\text{CHR}^{\vee}$ by allowing disjunctions in the rule bodies [3]. This extension makes it possible to perform search in CHR. Implementations of CHR(Prolog): CHR on top of Prolog already are implementations of $\text{CHR}^{\vee}$ as well. These implementations handle disjunctions by using the Prolog built-in (depth first) search mechanism.

Our aim is to implement different search strategies in CHR(Prolog). We start with a more detailed look at the context, motivation and goals of this paper

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

in Section 2. In Section 3, we extend the refined operational semantics of CHR towards $\text{CHR}^{\vee}$, supporting the definition of different search strategies. In Section 4, we further refine this semantics to make it more suitable for a trailing based implementation. Then, in Section 5, we propose a source to source transformation to implement breadth first search, making use of the K.U.Leuven CHR system [15] on top of SWI-Prolog [20]. We evaluate our approach in Section 6 and give an overview of related work in Section 7. Section 8 concludes.

2 Context, Motivation and Goals

We investigate how to implement different search strategies in the K.U.Leuven CHR system, running on top of SWI-Prolog. The main motivation is that we are developing a Constraint Logic Programming system, capable of handling nonlinear constraints over the real numbers, called INCLP($\mathbb{R}$) [13]. This system is implemented using the above mentioned CHR and Prolog implementations.

Search strategies are a fundamental part of CLP systems. A well chosen strategy can decrease the runtime considerably. Although CHR was originally designed for the implementation of constraint solvers, its limited ability to implement search algorithms, especially for CHR(Prolog) implementations, is a major weakness. We note that this limitation is mainly caused by the host language. Languages like Java that do not have a built-in search mechanism, need an explicit implementation of search and often offer more freedom with respect to search strategies. For example, the Java Constraint Kit [2] offers different search strategies by using its Java Abstract Search Engine module [14].

2.1 Breadth First Search

In this paper, we focus on the breadth first search strategy. This is a very basic strategy, but it exhibits two important challenges in implementing search strategies. Here, we take a closer look at these problems. Figure 1 depicts a search tree that is traversed in breadth first order. The nodes are visited in alphabetical order and the edges in numerical order. The edges 3 and 6 are edges that have been processed before, but need to be reapplied in order to be able to reach the next unvisited node. If this is done by recomputing them from scratch, we are essentially performing iterative deepening instead of breadth first search.

Non-standard Visiting Order When a choice point is created in Prolog, all its alternatives have to be tried before backtracking to a higher level. When visiting a node in breadth first order, the children of the current node are alternatives that are to be postponed until after all nodes at the current level are traversed. To implement such an order, we need a global representation of the choice points.

Revisiting Nodes Another difficulty is that in breadth first search, nodes of the search tree are revisited when going to the next level. It is important to note here that we do not want to repeat all the computations on the edge that terminated

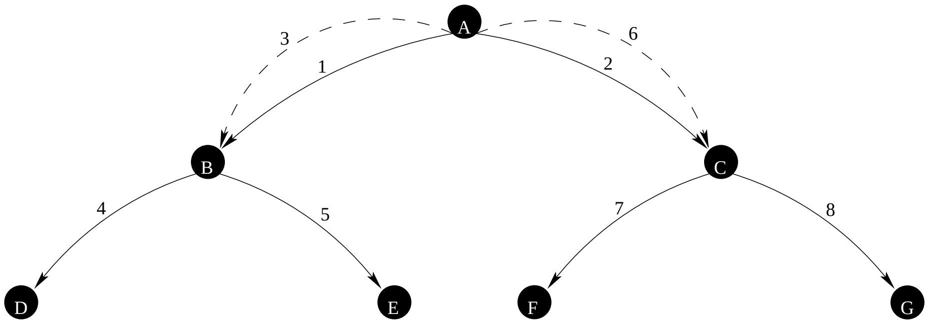

Figure1. Search Tree

in the repeated node. Instead, we wish to store the result and load this result when revisiting. In the context of CLP, nodes correspond to splitting the domain of a variable and the edges that connect the nodes represent the computations to return to a consistent state. Reaching a consistent state often takes a lot of time to compute, but quite limited space to store the result. For example, the INCLP($\mathbb{R}$) system uses a computationally quite expensive interval Newton iteration to achieve consistency, whereas the result only consists of some changed variable domains. Note that the ability to quickly move from one node to another already visited node, forms the main difference between breadth first search and iterative deepening.

2.2 Goals

The goals of this paper are the following:

- to define an extension of the refined operational semantics that supports disjunctions in the rule bodies and allows to define different search strategies
- to prove that it is possible to implement a search strategy different from depth first, by only using the language features of CHR and Prolog that are currently available
- to investigate what kind of extra support from the host language and the CHR system could make our task much easier and/or offer a better performance

3 A Combination of the Refined Operational Semantics ω_r and CHR$^\vee$

In this section, we briefly review the theoretical operational semantics ω_t [1] of CHR and how it is extended to become a semantics for CHR$^\vee$ [3]. We then review the refined operational semantics ω_r [6] and show how it can be extended in a similar way.

3.1 The Theoretical Operational Semantics ω_t and CHR$^\vee$

A CHR constraint c is an atom $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with a unique identifier i. We introduce the functions *chr* and *id*, defined as $chr(c\#i) = c$ and $id(c\#i) = i$. A CHR execution state is denoted by a tuple $\langle G, S, B, T\rangle_n$ where G is the *goal*, S the *CHR constraint store*, B the *built-in store*, T the *propagation history* and n the *next free identifier*. We assume familiarity with these concepts, see for example [6].

The operational semantics is described by a list of transitions that transform one CHR execution state into another one. If no more transitions are applicable, a *final* execution state has been reached. This is either a state with an empty goal: a successful final execution state, or a state with an inconsistent built-in store: a failed final execution state. The transitions of ω_t can be found in [6].

CHR$^\vee$ is an extension of CHR where disjunctions are allowed in the bodies of rules. A theoretical semantics $\omega_t^\vee$ for CHR$^\vee$ is presented as an extension of ω_t. We introduce a set of CHR execution states called the set of alternatives: $\bar{E} = \{E_1, \ldots, E_n\}$. The following transitions manipulate the set of alternatives and are an extension of [3]:

S1. Derive $\{\sigma\} \cup \bar{E} \rightarrowtail \{\sigma'\} \cup \bar{E}$ if there exists a transition $\sigma \rightarrowtail_{\omega_t} \sigma'$.
S2. Split $\{\langle (G_1 \vee G_2) \wedge G, S, B, T\rangle_n\} \cup \bar{E} \rightarrowtail \{\langle G_1 \wedge G, S, B, T\rangle_n, \langle G_2 \wedge G, S, B, T\rangle_n\} \cup \bar{E}$.

The **Derive** transition supports both processing states in parallel or sequentially in any order. It is desirable to have more control over the search order, but since the theoretical operational semantics is already highly nondeterministic, it is not very meaningful to impose a particular search order on it.

3.2 The Refined Operational Semantics ω_r

The refined operational semantics ω_r is an implementation of the theoretical operational semantics that makes the execution considerably more deterministic and is more closely related to actual implementations of CHR.

A CHR execution state is a tuple $\langle A, S, B, T\rangle_n$ where A is the *execution stack* and S, B, T and n are as in ω_t. The transitions of ω_r can be found in [6].

3.3 A Refined Operational Semantics for CHR$^\vee$

In this subsection, we introduce the $\omega_r^\vee$ semantics for CHR$^\vee$ which extends the refined operational semantics of CHR. This extension supports the definition of search strategies, which is not present in the theoretical semantics $\omega_t^\vee$.

We make the set of alternatives of $\omega_t^\vee$ more concrete by implementing it as an ordered sequence of CHR execution states called the list of alternatives: $\bar{E} = [E_1, E_2, \ldots, E_n]$ where the state E_1 is the active execution state and $E_2, \ldots, E_n$ are the remaining alternatives. We propose the following transitions that manipulate the list of alternatives:

S1. Derive $[\sigma \mid \bar{E}] \rightarrowtail [\sigma' \mid \bar{E}]$ if there exists a transition $\sigma \rightarrowtail_{\omega_r} \sigma'$.

S2a. Split (Depth First) $[\sigma \mid \bar{E}] \rightarrowtail [\sigma_1, \ldots, \sigma_m \mid \bar{E}]$ where $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$ for $1 \leq i \leq m$. This transition implements a depth first search.

S2b. Split (Breadth First) $[\sigma \mid \bar{E}] \rightarrowtail \bar{E} +\!\!+ [\sigma_1, \ldots, \sigma_m]$ where $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$ for $1 \leq i \leq m$. This transition implements a breadth first search.

S3. Next $[\sigma \mid \bar{E}] \rightarrowtail \bar{E}$ if σ is final CHR execution state. This transition is applied automatically if E is a failed final state, but requires an external event (explicit call for the next solution, e.g. from the toplevel) if E is a successful final state.

The given **Split** transitions are only two of the possibilities. Other search strategies can be implemented easily using similar transitions. For example, best first search can be implemented by sorting the list of remaining alternatives according to some heuristic. Strategies like intelligent backtracking can be implemented by removing states from the remaining alternatives. A branch and bound algorithm can be implemented by adding the provisional optimum as a constraint to the remaining alternatives.

We require that at the moment of a **Next** transition, it is known which of the alternatives is to be chosen next. This implies that it is not always necessary to have a full order between the different alternatives and allows implementing more dynamic search strategies.

While the **Split** transition in $\omega_t^\vee$ only supports binary disjunctions, the $\omega_r^\vee$ **Split** transitions supports n-ary disjunctions. The reason for this generalization is that although n-ary disjunctions can be logically modeled as a series of nested binary disjunctions, they do not behave equivalently with respect to all search strategies.

4 The Tree-based Operational Semantics $\boldsymbol{\omega}_\curlywedge$

In the above presentation, the **Split** transitions create copies of the CHR constraint store, the built-in store and the propagation history for each of the alternatives. These copies can then be changed independently by the ω_r transitions. In practice copying the execution state is often very expensive and should be avoided if possible. It is also hard to implement in our current CHR implementation. Another issue is that when changing the active execution state by using the **Split** or **Next** transitions, we have ignored the fact that these execution states are often largely the same. These considerations influence implementations of the proposed semantics.

In this section, we propose the tree-based operational semantics $\omega_\curlywedge$[1] which is a refinement of the $\omega_r^\vee$ semantics. It is based on the concept of a search tree and is more suitable as a basis for a practical implementation based on trailing. It makes the presentation in Section 5 more straightforward.

[1] Read as "omega tree".

4.1 Nodes and Edges

A search tree consists of a set of nodes and a set of (directed) edges connecting these nodes. A node is either an *internal* node or a *leaf* node. An internal node represents a choice point; the *root* node is a special internal node that starts the whole search process. A leaf node represents a successful or failed final execution state. An edge goes from one node, its *start* node, to another node, its *end* node, and represents the derivation that transforms one of the alternatives of its start node into its end node. Edges come in two flavors: *processed* edges and *unprocessed* edges. For a processed edge the derivation from start to end node has already been computed, and for unprocessed edges it has not yet. We now introduce a mapping between these concepts and CHR execution states.

Nodes are represented as CHR execution states. An internal node is represented by the state $\langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T\rangle_n$ where $m \geq 2$. The root node of the search tree is the initial state $\langle G, \emptyset, true, \emptyset\rangle_1$ with G the initial goal. A leaf node is simply a final state.

An unprocessed edge is represented by the CHR execution state $N = \langle [A_i \mid A], S, B, T\rangle_n$. The edge is processed by starting a derivation in that state. Such a derivation ends either in a successful or failed final CHR execution state (a leaf node) or in another choice point (an internal node). A processed edge connects its start node N with its end node N' and is represented as $N \frown N'$.

A search tree state is a tuple $\langle \sigma, N, \bar{E}_u, \bar{E}_p\rangle$. Here, σ is the current CHR execution state of the active edge or $\sigma = \epsilon$ if no edge is active. N is the current start node, which is also the start node of the active edge if an edge is active. $\bar{E}_u$ is an ordered sequence of (inactive) unprocessed edges and $\bar{E}_p$ is a set of processed edges. The edges in $\bar{E}_p$ form the part of the search tree that has already been explored and the edges in $\bar{E}_u$ form the boundary between the explored part and the unexplored part of the search tree. The initial search tree state is the tuple $\langle\langle G, \emptyset, true, \emptyset\rangle_1, \langle G, \emptyset, true, \emptyset\rangle_1, \epsilon, \emptyset\rangle$ The following transitions manipulate search tree states:

S1. Derive $\langle \sigma, N, \bar{E}_u, \bar{E}_p\rangle \rightarrowtail \langle \sigma', N, \bar{E}_u, \bar{E}_p\rangle$ if there is a transition $\sigma \rightarrowtail_{\omega_r} \sigma'$.

S2a. Split (Depth First) $\langle \sigma, N, \bar{E}_u, \bar{E}_p\rangle \rightarrowtail \langle \epsilon, \sigma, [\sigma_1, \ldots, \sigma_m \mid \bar{E}_u], \bar{E}_p \cup \{N \frown \sigma\}\rangle$ where $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T\rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T\rangle_n$ for $i \in 1 \ldots m$. This transition implements depth first search. The choice point creates a new node σ which becomes the current start node. The edge $N \frown \sigma$ is added to the set of processed edges. For every alternative of the choice point, a new unprocessed edge, starting in the node σ, is added in front of the list of unprocessed edges.

S2b. Split (Breadth First) $\langle \sigma, N, \bar{E}_u, \bar{E}_p\rangle \rightarrowtail \langle \epsilon, \sigma, \bar{E}_u +\!\!+ [\sigma_1, \ldots, \sigma_m \mid \bar{E}_u], \bar{E}_p \cup \{N \frown \sigma\}\rangle$ where $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T\rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T\rangle_n$ for $i \in 1 \ldots m$. This transition implements breadth first search. The choice point creates a new node σ which becomes the current start node. The edge $N \frown \sigma$ is added to the set of processed edges. For every alternative of the choice point, a new unprocessed edge, starting in the node σ, is added to the back of the list of unprocessed edges.

S3. Next $\langle \epsilon, N, [\sigma_i \mid \bar{E}_u], \bar{E}_p \rangle \rightarrowtail \langle \sigma_i, N, \bar{E}_u, \bar{E}_p \rangle$ where $N = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$. This transition activates the next unprocessed edge. It requires that the current start node corresponds to the start node of the next unprocessed edge.

S4. Move Down $\langle \epsilon, N, [\sigma_i \mid \bar{E}_u], \bar{E}_p \cup \{N \frown N'\} \rangle \rightarrowtail \langle \epsilon, N', [\sigma_i \mid \bar{E}_u], \bar{E}_p \cup \{N \frown N'\} \rangle$ if node N is an ancestor of N'', the start node of σ_i, and the edge $N \frown N'$ is on the path between nodes N and N'' or $N' = N''$

S5a. Move Up (Internal Node) $\langle \epsilon, N, [\sigma_i \mid \bar{E}_u], \bar{E}_p \cup \{N' \frown N\} \rangle \rightarrowtail \langle \epsilon, N', [\sigma_i \mid \bar{E}_u], \bar{E}_p \cup \{N' \frown N\} \rangle$ if node N is not an ancestor node of the start node of σ_i.

S5b. Move Up (Successful Leaf Node) $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle \rightarrowtail \langle \epsilon, N, \bar{E}_u, \bar{E}_p \rangle$ if σ is a successful final state, a solution. To be applied, this transition requires an external event asking for the next solution.

S5c. Move Up (Failed Leaf Node) $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle \rightarrowtail \langle \epsilon, N, \bar{E}_u, \bar{E}_p \rangle$ if the execution state σ is a failed final CHR execution state.

4.2 Equivalence of the $\omega_\curlywedge$ and $\omega_r^\vee$ semantics

We can show that the $\omega_\curlywedge$ semantics presented in this section, is operationally equivalent to the $\omega_r^\vee$ semantics presented in Section 3. For this purpose, we introduce the following abstraction function:

Definition 1. *We define an abstraction function α that maps a search tree state on a list of alternatives as follows:*

$$\alpha(\langle \langle A, S, B, T \rangle_n, \_, \bar{E}_u, \_ \rangle) = [\langle A, S, B, T \rangle_n \mid \bar{E}_u]$$
$$\alpha(\langle \epsilon, \_, \bar{E}_u, \_ \rangle) = \bar{E}_u$$

The equivalence proof consists of the following theorems:

Theorem 1. *For every two lists of alternatives $\bar{E}_1$ and $\bar{E}_2$ for which holds that $\bar{E}_1 \rightarrowtail_{\omega_r^\vee} \bar{E}_2$, there exist two search tree states S_1 and S_2 for which a derivation $S_1 \rightarrowtail^*_{\omega_\curlywedge} S_2$ exists such that $\alpha(S_1) = \bar{E}_1$ and $\alpha(S_2) = \bar{E}_2$.*

Theorem 2. *For every two search tree states S_1 and S_2 for which holds that $S_1 \rightarrowtail_{\omega_\curlywedge} S_2$, we have that either $\alpha(S_1) = \alpha(S_2)$ or there exist two lists of alternatives $\bar{E}_1$ and $\bar{E}_2$ for which a transition $\bar{E}_1 \rightarrowtail_{\omega_r^\vee} \bar{E}_2$ exists such that $\alpha(S_1) = \bar{E}_1$ and $\alpha(S_2) = \bar{E}_2$.*

The proofs of these theorems can be found in [12].

5 Implementation

In this section we give an overview of how a breadth first search strategy can be implemented for the K.U.Leuven CHR system [15] in SWI-Prolog [20] using a source to source transformation. Apart from some practical issues that are implementation specific, most of the ideas that are presented here can be generalized to other CHR and Prolog implementations and even to CHR implementations not running on top of Prolog.

5.1 Nodes and Edges

In Section 4 we have introduced the concepts of nodes and edges. We now present how we can implement the processing of edges. The next unprocessed edge can only become active if the current start node is also the start node of the next edge. If the current start node is a different node, it first needs to be changed: if it is an ancestor of the next edge's start node, there exists an already processed edge on the path between the current start node and the next edge's start node.[2] The changes made by the processed edge are loaded and its end node becomes the current start node. This process is repeated until the current start node is equal to the start node of the next unprocessed edge. Otherwise, if the current start node is not an ancestor of the next edge's start node, the changes on the edge of which the current start node is the end node, are backtracked and the current start node is changed to that edge's start node. This process is repeated until the current start node is an ancestor of the next edge's start node.

For the implementation it is advantageous to also allow already processed edges to become active. This way, we can treat processed and unprocessed edges in a similar way. Instead of initiating a derivation, the activation of a processed edge only causes the current start node to be changed to its end node. When transferring to the next edge in the list of unprocessed edges, processed edges that are on the 'path' between the current start node and the next unprocessed edge's start node, temporarily become active so as to change the current start node.

```
activate_next_edge, active_edge(Edge) <=>
        next_edge(NextEdge),
        same_branch(Edge,NextEdge),
        (   edge_end_node(Edge,Node),
            edge_start_node(NextEdge,Node)
        ->  set_next_edge,
            active_edge(NextEdge)
        ;   next_on_path(Edge,NextEdge,Between),
            active_edge(Between)
        ).
```

The `next_edge/1` predicate unifies the next scheduled edge with its argument. The `same_branch/2` predicate checks whether the two given edges are on a single branch in the search tree. The combination of both predicates must be tried again on backtracking, when an edge of the list of unprocessed edges has been processed (this is denoted by a call to `set_next_edge/0`). If the call to `same_branch/2` fails, the changes made by the current active edge are undone. The Prolog if-then-else (`->` `;`) is used to cut away unnecessary choice points. The `next_on_path/3` predicate unifies its last argument with the first edge that lies on the path between the edges given by its first two arguments.

To handle an active edge, we use the following rules, distinguishing between unprocessed and processed edges:

[2] If it is not already processed, the next edge's start node cannot exist.

```
active_edge(Edge) ==>
        edge_status(Edge,unprocessed) |
        edge_goal(Edge,Goal),
        call(Goal),
        (   true
        ;   activate_next_edge
        ).
active_edge(Edge) ==>
        edge_status(Edge,processed) |
        load_edge_changes(Edge),
        activate_next_edge.
```

The first rule handles an unprocessed edge. Its goal is called which causes a derivation that either ends in the creation of an end node (**Split** transition) in which case the next edge is activated automatically, or in a final execution state. If it ends in a successful final state, the next edge is activated on request (solution). We use the Prolog disjunction (`;`) for this purpose: the disjunctives are traversed left-to-right and depth first. If it ends in a failed final state, the derivation is backtracked automatically.

The second rule handles an already processed edge. For such an edge, the changes are loaded and it is again tried to activate the next unprocessed edge.

5.2 State Changes

An edge represents a derivation that connects its start node with its end node. For our purposes, it is sufficient to only look at edges that end in a choice point (i.e. an internal node). In [12], we show that every derivation can be written as $\langle A, S \cup S_-, B, T\rangle_n \rightarrowtail^*_{\omega_r} \langle A', S \cup S_+, B \wedge B_+, T \cup T_+\rangle_{n+n_+}$. Here the sets S_+ and S_- contain respectively the added and removed CHR constraints. The conjunction B_+ contains the added built-in constraints, T_+ represents the new propagation history tuples and n_+ is the increase of the next free identifier number. Finally, the derivation changes the activation stack A into the new activation stack A'.

For the particular case of the derivation of an edge, we have $\langle [A_i \mid A], S \cup S_-, B, T\rangle_n \rightarrowtail^*_{\omega_r} \langle [A'_1 \vee \ldots \vee A'_m \mid A'], S \cup S_+, B \wedge B_+, T \cup T_+\rangle_{n+n_+}$, where $[A_i \mid A]$ is one of the alternatives of the choice point represented by the edge's start node.

To be able to reconstruct this derivation, or in other words, to be able to move from one node to another, we need to be able to store how the CHR execution state is changed (i.e. create an explicit trail). We do not need the activation stack of the end node: the new unprocessed edges that start at this node (as created by a **Split** transition) already contain the information we need. Therefore, we only need to have a representation for the sets S_+, S_- and T_+, the conjunction B_+ and the integer n_+.

Our approach is similar to the explicit trailing mechanism of [16], but extends it by not only collecting changes to the CHR constraint store, but also to the built-in constraint store and the propagation history. We now describe how we can collect, store and load the changes.

Explicit Identifiers and Propagation History The identifiers of CHR constraints and the propagation history are not available to the CHR programmer. They are needed to create the sets S_-, S_+ and T_+, and to find the integer n_+. It is easy to add an explicit identifier to every constraint and to make the propagation history explicit. We show that this does not change the operational behavior of a CHR program in [12].

Changes to the CHR Constraint Store The CHR constraint store can change by adding identified constraints to it and by removing identified constraints from it. We show how these changes can be collected and stored and how we can load the stored changes. We use explicit identifiers for this purpose.

For every constraint `c/n` we create three new constraints: an *id-extended* constraint: `c_id/n+1`, a constraint to denote a constraint addition: `c_add/n+1` and a constraint to denote a constraint deletion: `c_del_n/1`. We add an explicit identifier and create a new element in S_+ by the following kind of rules:

```
c(X̄) <=>
        new_id(ID),
        c_add(ID,X̄),
        c_id(ID,X̄).
```

Here `new_id/1` creates a new unique identifier. Every simplification and simpagation rule of the form

```
r @ c₁(X̄₁), ..., cᵢ(X̄ᵢ) \
    cᵢ₊₁(X̄ᵢ₊₁), ..., cₙ(X̄ₙ) <=>
        guard |
        body.
```

is changed into a rule

```
r @ normal, c_id₁(ID₁,X̄₁), ..., c_idᵢ(IDᵢ,X̄ᵢ) \
    c_idᵢ₊₁(IDᵢ₊₁,X̄ᵢ₊₁), ..., c_idₙ(IDₙ,X̄ₙ) <=>
        guard |
        c_del_nᵢ₊₁(IDᵢ₊₁),
        ...,
        c_del_nₙ(IDₙ),
        prelink(t(X̄₁,...,X̄ₙ)),
        body.
```

The `normal` constraint is used to turn rules on and off so that during the **Move Up** and **Move Down** transitions, no rules can fire, making these transitions behave as an atomic operation. For every removed constraint, an element of S_- is created. The use of the `prelink/1` predicate is explained further on, in the paragraph about the built-in constraint store.

Constraints that are added to and removed from the store on a single edge, must be removed from S_+. Therefore, for every constraint `c/n`, we create a rule:

```
c_del_n(ID), c_add(ID,_X1,...,_Xn) <=> true.
```

We can store the changes by collecting them in a list and saving the list in a non-backtrackable way.

Changes to the Propagation History Every propagation rule of the form

```
r @ c₁(X̄₁), ..., cₙ(X̄ₙ) ==>
        guard |
        body.
```

is converted into a rule

```
r @ normal, c_id₁(ID₁,X̄₁), ..., c_idₙ(IDₙ,X̄ₙ) ==>
        \+ in_history(t(ID₁, ..., IDₙ, r)),
        guard |
        add_to_history(t(ID₁,...,IDₙ,r)),
        prelink(t(X̄₁,...,X̄ₙ)),
        body.
```

Because the propagation history is now directly available to us, it is easy to store how it is changed.

Changes to the Built-in Constraint Store For the built-in constraint store, we restrict ourselves to pure Prolog. In particular, we exclude features like attributed variables or global variables. Since the CHR implementation on which this work is based, is implemented using these features, we cannot support these features without losing the distinction between the CHR constraints and the built-in constraints.

In pure Prolog, the built-in store consists of a set of variable bindings, which can be represented as a list $[x_1 = t_1, \ldots, x_n = t_n]$ with x_i variables and t_i terms for $1 \leq i \leq n$. We need to be able to reconstruct the variable bindings for all relevant variables that appear in the derivation of the active edge. The relevant variables are the ones that are not *strictly local*, where strictly local variables are defined as the variables that do not occur in the initial goal, nor in any of the CHR constraints [1].

In the implementation, we construct two lists: one containing the variables $[x_1, \ldots, x_n]$ and one containing the terms $[t_1, \ldots, t_n]$. When a new variable appears, it is added to the list of variables. When finishing processing the active edge (because of a **Split** transition), this list of variables, which meanwhile has changed into a list of terms, is copied. On backtracking, the bindings of the variables are undone in the original list, but not in the copy.[3] This gives us the two lists that we need: the original list is the list of variables and the copy is the list of terms. The bindings can be reapplied by unifying the two lists.

The `prelink/1` predicate that we used in the transformation of the original program rules, adds the new variables in the term that is its argument, to the original list of variables. The transformation guarantees that all relevant variables are stored, because all variables that occur in the CHR constraint store and are bound, must be in the head of some rule.

[3] This is related to the way terms are constructed.

6 Evaluation

The implementation that has been presented in the previous section, has been used to transform some small example programs so that they are executed using depth first or breadth first search (i.e. both versions of the **Split** transition have been implemented).

6.1 Benchmarks

The following benchmark is performed on a 2.8 GHz Pentium IV processor using SWI-Prolog version 5.6.0. It forms a first indication of what is possible without having extra built-in support from the host language. A more fine-tuned implementation can probably still decrease the runtimes somewhat.

The benchmark consists of finding all solutions of a Sudoku puzzle with 295 different solutions. We measure the runtime, generated garbage and global stack (heap) space required for storing the search tree after the first solution is found. The search tree in this benchmark consists of 8 143 nodes and has a depth of 57.

Search Strategy	time(s)	garbage(MB)	global stack(MB)
Breadth First	50.14	420.82	2.22
Depth First (Explicit)	6.12	47.92	0.24
Depth First (Implicit)	2.04	1.08	0.12

Timings for some instances of the n queens problem can be found in [12]. The explicit version of depth first is using our program transformation, while the implicit version is using the built-in Prolog depth first search mechanism. The different timings between the explicit and implicit versions of depth first, show that there is a considerable overhead by making search explicit. This overhead is caused by the extra tasks of maintaining and storing an explicit trail and by making the propagation history and choice points explicit. The extra overhead of breadth first search is mostly related to loading the stored trails and in particular updating the internal hash tables. In this benchmark, these hash tables are responsible for at least a third of the runtime.

The amount of garbage generated is considerably larger in the explicit search strategies compared to implicit depth first. Moreover, it is much larger for breadth first than for depth first. This is explained by the fact that breadth first uses much more **Move Up** transitions: in the benchmark 113 391 times compared to 8 084 times for depth first.

Since all solutions of this benchmark are at the same depth, the part of the search tree that has to be in memory is maximal when reaching the first solution. This holds both for explicit depth first and explicit breadth first. Breadth first requires much more memory as all lower depth nodes that are potentially part of a full solution, have been visited and are in memory.

Finally, the benchmark clearly does not present breadth first search as a better alternative for depth first search. Although it is easy to find examples on which breadth first performs arbitrarily better than depth first, our aim is showing what kind of overhead is generated by explicitly programming a different search strategy in CHR. The choice of good search strategies for various constraint programming problems is a completely different discussion.

6.2 Support from the Host Language

For different search strategies to become practical, they have to be implemented efficiently and it is clear that our source transformation is not highly performant. We can thus only conclude that extra support from the host language is necessary. In this subsection, we give our view of what is needed.

The main issue is to be able to jump from one node to another efficiently and to be able to return to previously visited nodes. We have implemented an explicit trailing mechanism for this, but such a mechanism can also be implemented at a lower level. An example of this is the XSB system [19] which supports tabling. Its SLG-WAM engine allows undoing changes on the trail without losing the information in the trail. Redoing changes is done by traversing the trail in a so-called forward mode. It also creates an explicit choice point stack. Instead of using an explicit trail, tabling can also be implemented using copying [5].

We think it is useful from a flexibility point of view, to have an explicit representation of choice points and low level primitives to move from one choice point to another. This supports more freedom to implement special search strategies. It might even be useful to allow the programmer to specify how the changes between different choice points should be stored: by copying, by trailing or not at all (recomputation). Finally, we note that host language support for the specification of different search strategies, can be easily extended to CHR implementations that are compiled into this host language, like CHR(Prolog).

7 Related Work

Adding different search strategies to declarative languages and in particular Logic Programming languages, has been done before. In [10], an operator for encapsulating search in the functional logic programming language Curry is presented. The relation with Prolog predicates like `findall/3` is given, noting that such predicates can return the solutions in any order if there are only finitely many, but cannot really search in a different than depth first order. The implementation of the search operator relies on a variable scoping mechanism, which is similar to having multiple copies of the variables. A similar idea is presented in Oz [18]. This language has a form of constraint store called a blackboard. The search operator creates local versions of this blackboard, with copies of the constraints that are on the global blackboard.

In [17] copying is combined with recomputation and compared with trailing. The Java Abstract Search Engine (JASE) which is part of the Java Constraint Kit (JACK) [2] expands on these ideas. It supports different search strategies amongst which breadth first search and restores states by using trailing, copying or recomputation, or by using a combination of these.

In Ciao Prolog [11], breadth first and iterative deepening search are supported using a source transformation. The breadth first implementation is based on the `findall/4` predicate,[4] which enumerates all alternatives, making copies of the variables. By using a limited form of interpretation, the alternative bodies of breadth first predicates are added to a list of alternatives, while for depth first predicates, all alternative solutions are added. This only works if the depth first predicate has only finitely many solutions.

Finally, in [4], the blackboard primitives of SICStus Prolog are used to implement intelligent backtracking in Prolog.

8 Conclusions and Future Work

We have formalized a theoretical semantics $\omega_t^\vee$ for CHR$^\vee$: CHR extended with disjunctions in rule bodies. We have introduced $\omega_r^\vee$, an extension of the refined operational semantics [6] of CHR towards CHR$^\vee$. This $\omega_r^\vee$ semantics supports the definition of different search strategies. These results extend the work in [3].

We refined our $\omega_r^\vee$ into an operationally equivalent formulation that is more suitable as a basis for trailing based implementations. We described how a breadth first search strategy can be implemented as a source to source transformation, using the currently available language features of the K.U.Leuven CHR system [15] and SWI-Prolog [20] only. This implementation introduces an explicit trailing mechanism that is an extension of the one in [16]. A first evaluation of the implementation has shown that the overhead created by the transformation is considerable and we have made suggestions to what kind of low level support from the host language could help to improve the performance.

Future Work Future work consists of adding low level support for search to the host language and the creation of a practical declarative framework for the specification of search strategies in CHR programs. We think this is essential if we want to use CHR to write highly performant CLP systems.

References

1. Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *CP'97: Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, pages 252–266, Schloss Hagenberg, Austria, 1997. Springer Verlag.
2. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A Java constraint kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming, Kiel*, Kiel, Germany, September 2001.
3. Slim Abdennadher and Heribert Schütz. CHR$^\vee$: 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.

[4] A difference list version of the more well-known `findall/3` predicate.

4. Maurice Bruynooghe. Enhancing a search algorithm to perform intelligent backtracking. *TPLP*, 4(3):371–380, 2004.
5. Bart Demoen and Konstantinos F. Sagonas. CAT: The copying approach to tabling. *Journal of Functional and Logic Programming*, 1999(Special Issue 2), 1999.
6. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, St-Malo, France, Sep 2004. Springer Verlag.
7. Michael Fink, Hans Tompits, and Stefan Woltran, editors. *Proceedings of the 20th Workshop on Logic Programming*. INFSYS Research Report 1843-06-02 (TU Wien), 2006.
8. Thom W. Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 1994.
9. Michael Hanus. Adding Constraint Handling Rules to Curry. In Fink et al. [7], pages 81–90.
10. Michael Hanus and Frank Steiner. Controlling search in declarative programs. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *PLILP/ALP*, volume 1490 of *Lecture Notes in Computer Science*, pages 374–390. Springer, 1998.
11. Manuel V. Hermenegildo, Francisco Bueno, Daniel Cabeza, Manuel Carro, Maria J. García de la Banda, Pedro López-García, and Germán Puebla. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems. In Paqui Lucio, Maurizio Martelli, and Marisa Navarro, editors, *APPIA-GULP-PRODE*, pages 105–110, 1996.
12. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. Flexible search strategies for Prolog CHR. Technical Report CW 447, K.U.Leuven, Dept. CS, May 2006.
13. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. INCLP(R) - Interval-based nonlinear constraint logic programming over the reals. In Fink et al. [7], pages 91–100.
14. Ekkehard Krämer. A generic search engine for a Java constraint kit. Master's thesis, Institute of Computer Science, LMU, Munich, Germany, January 2001.
15. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In *First workshop on constraint handling rules: selected contributions*, pages 1–5, 2004.
16. Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. Automatic implication checking for CHR constraints. In Horatiu Cirstea and Narciso Martí-Oliet, editors, *RULE'05: Proceedings of the 6th International Workshop on Rule-Based Programming*, Nara, Japan, April 2005.
17. Christian Schulte. Comparing trailing and copying for constraint programming. In *ICLP*, pages 275–289, 1999.
18. Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In Alan Borning, editor, *PPCP*, volume 874 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 1994.
19. Terrance Swift and David Scott Warren. An abstract machine for SLG resulution: Definite programs. In *SLP*, pages 633–652, 1994.
20. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, December 2003.