Flexible Search Strategies in Prolog CHR

Leslie De Koninck
Tom Schrijvers
Bart Demoen

Report CW 447, May 2006



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Flexible Search Strategies in Prolog CHR

Leslie De Koninck
Tom Schrijvers
Bart Demoen

Report CW 447, May 2006

Department of Computer Science, K.U.Leuven

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 with Prolog as a 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 on some benchmarks and give directions for future work.

Keywords : Constraint Handling Rules, Prolog, search strategies, semantics, implementation.
CR Subject Classification : D.1.6 [Programming Techniques] Logic Programming, D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages, D.3.3 [Programming Languages] Language Constructs and Features — Control structures

Flexible Search Strategies in Prolog CHR

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 with Prolog as a 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 on some benchmarks 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 language.

Constraint Handling Rules [8] is a high-level rule-based language, built on top of a host language like Prolog [14], Java [2,19], Haskell or Curry [9], and designed for a more easy implementation of Constraint Programming facilities. CHR can be extended to CHR$^\vee$ by allowing disjunctions in the rule bodies [3]. This extension makes it possible to perform search in CHR. Implementations of CHR on top of Prolog already are implementations of 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$^\vee$ on top of Prolog. We start with a more detailed look at the context, motivation and goals of this paper in Section 2. In Section 3, we extend the refined operational semantics of CHR towards 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 [14] 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 presents our conclusions.

* 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).

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($\mathbb{R}$) [12]. 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. Consider for example the problem of minimizing an objective function in a constrained search space. For nonlinear constraints, often a branch and bound approach is the only alternative. Using interval arithmetic, we can find safe bounds for the value of the objective function and by probing random points that satisfy the constraints, we can find achievable objective function values. If the objective function bounds in one branch are not better than the achievable objective function value in another branch, it makes no sense to look at the first branch any further. Using a standard left to right depth first search, we risk doing a lot of unnecessary computations if the leftmost branch is suboptimal.

Although CHR was originally designed for the implementation of constraint solvers, its limited ability to implement search algorithms, especially for systems running on top of Prolog, 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 (JACK) [2] offers different search strategies by using its Java Abstract Search Engine module (JASE) [13].

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.

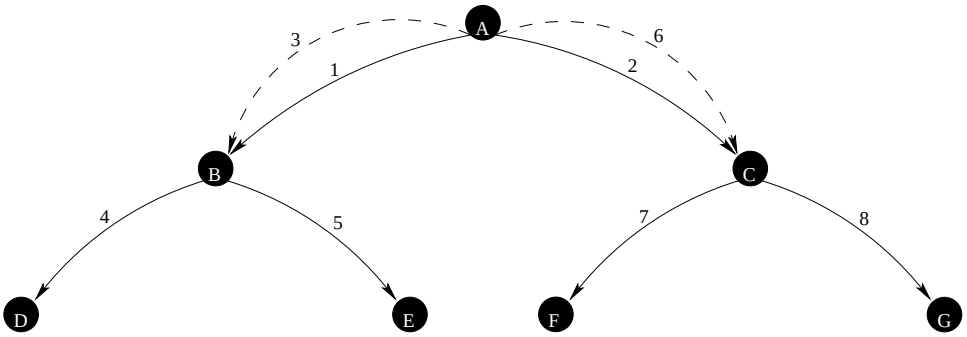

Figure1. Search Tree

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 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_i are terms for $1 \leq i \leq n$. We call predicate symbol p with arity n the predicate of c. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with a unique identifier i. We introduce the functions chr and id which are defined as $chr(c\#i) = c$ and $id(c\#i) = i$. A CHR execution state $\langle G, S, B, T \rangle_n$ consists of the following elements:

- The *goal* G is a multiset of constraints to be executed (both built-in and CHR)
- The *CHR constraint store* S is a set of identified CHR constraints (their identifiers make them unique)
- The *built-in constraint store* B is a conjunction of built-in constraints
- The *propagation history* T is a set of tuples that is used to prevent a rule from firing more than once for the same combination of constraints.

– The *next free identifier* n is an integer that is used when giving unique identifiers to CHR constraints.

The operational semantics as a state transition system. 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 are shown in Table 1. Here, $\mathcal{D}$ denotes the constraint domain of the built-in constraints and $\bar{\exists}_A F = \exists X_1, \ldots, \exists X_n F$ with $\{X_1, \ldots, X_n\} = vars(F) - vars(A)$.

1. Solve $\langle \{c\} \uplus G, S, B, T \rangle_n \rightarrowtail \langle G, S, c \wedge B, T \rangle_n$ where c is a built-in constraint.
2. Introduce $\langle \{c\} \uplus G, S, B, T \rangle_n \rightarrowtail \langle G, \{c\#n\} \cup S, B, T \rangle_{n+1}$ where c is a CHR constraint.
3. Apply $\langle G, H_1 \cup H_2 \cup S, B, T \rangle_n \rightarrowtail \langle C \uplus G, H_1 \cup S, \theta \wedge B, T' \rangle_n$ where there exists a (renamed apart) rule in P of the form

$$r @ H_1' \backslash H_2' \iff g \mid C$$

and a matching substitution θ such that $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$ and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$ and the tuple $id(H_1) +\!\!+ id(H_2) +\!\!+ [r] \notin T$. In the result $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [r]\}$.

Table1. Transitions of the Theoretical Operational Semantics ω_t

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.

For each constraint predicate p, we number the occurrences of p in the different CHR rules from top to bottom and from right to left. An *occurrenced* identified constraint $c\#i : j$ is the identified constraint $c\#i$ that is being considered at the j-th occurrence of its predicate.

A CHR *execution state* is a tuple $\langle A, S, B, T \rangle_n$. Here A is the *execution stack*: a sequence of built-in constraints, identified CHR constraints and occurrenced identified CHR constraints. The first element of the execution stack is called the active constraint. The CHR constraint store S, the built-in constraint store B, the propagation history T and the next free identifier n are the same as in ω_t. The transitions of ω_r are shown in Table 2.

1. **Solve** $\langle [c \mid A], S_0 \cup S_1, B, T\rangle_n \rightarrowtail \langle S_1 +\!\!+ A, S_0 \cup S_1, c \wedge B, T\rangle_n$ where c is a built-in constraint, and $vars(S_0) \subseteq fixed(B)$ is the set of variables fixed by B. This reconsiders constraints whose matches might be affected by c.
2. **Activate** $\langle [c \mid A], S, B, T\rangle_n \rightarrowtail \langle [c\#n : 1 \mid A], \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. **Reactivate** $\langle [c\#i \mid A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : 1 \mid A], S, B, T\rangle_n$ where c is a CHR constraint.
4. **Drop** $\langle [c\#i : j \mid A], S, B, T\rangle_n \rightarrowtail \langle A, S, B, T\rangle_n$ where $c\#i : j$ is an occurrenced active constraint and there is no such occurrence j in P.
5. **Simplify** $\langle [c\#i : j \mid A], \{c\#i\} \cup H_1 \cup H_2 \cup H_3 \cup S, B, T\rangle_n \rightarrowtail \langle C +\!\!+ A, H_1 \cup S, \theta \wedge B, T\rangle_n$ where the jth occurrence of the predicate of c in a (renamed apart) rule in P is
$$r@H_1' \backslash H_2', d_j, H_3' \iff g \mid C$$
and there exists a matching substitution θ such that $c = \theta(d_j)$, $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $chr(H_3) = \theta(H_3')$ and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$.
6. **Propagate** $\langle [c\#i : j \mid A], \{c\#i\} \cup H_1 \cup H_2 \cup H_3 \cup S, B, T\rangle_n \rightarrowtail \langle C +\!\!+ A, H_1 \cup S, \theta \wedge B, T'\rangle_n$ where the jth occurrence of the predicate of c in a (renamed apart) rule in P is
$$r@H_1', d_j, H_2' \backslash H_3' \iff g \mid C$$
and there exists a matching substitution θ such that $c = \theta(d_j)$, $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $chr(H_3) = \theta(H_3')$ and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$. If $H_3 = \emptyset$ it must also hold that the tuple $id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ [r] \notin T$. In the result, $T' = T \cup \{id(H_1 +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ [r]\}$. Otherwise, if $H_3 \neq \emptyset$, $T' = T$.
7. **Default** $\langle [c\#i : j \mid A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : j+1 \mid A], S, B, T\rangle_n$ if the current state cannot fire any other transition.

Table2. Transitions of the Refined Operational Semantics ω_r

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

In this subsection, we introduce $\omega_r^\vee$: an extension of the refined operational semantics of CHR and a semantics for CHR$^\vee$. 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.

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 $\omega_\curlywedge$

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.

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.

A node is represented by a CHR execution state. 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. It is the only internal node that may have only one alternative. A leaf node is simply a final state.

An unprocessed edge is represented by the CHR execution state $\sigma_i = \langle [A_i \mid A], S, B, T\rangle_n$. 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 $N = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T\rangle_n$ 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:

[1] Read as "omega tree".

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 exists 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) $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle \rightarrowtail \langle \epsilon, N, \bar{E}_u, \bar{E}_p \rangle$ where 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 the ω_λ and $\omega_r^\vee$ Semantics

In this subsection, we show that the ω_λ 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:*

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

Now to prove operational equivalence, we need the following results:

Lemma 1. *For every unprocessed edge σ_i in $\bar{E}_u$ with start node N, there exists a processed edge $N' \frown N$ in $\bar{E}_p$.*

Proof. Initially, $\bar{E}_u$ is empty. The only transitions that add edges to $\bar{E}_u$ are the **Split** transitions. These transitions create a processed edge $N' \frown N$ in $\bar{E}_p$ with N the start node of the new unprocessed edges in $\bar{E}_u$. □

Lemma 2. *The current start node appears in $\bar{E}_p$ or is the root node.*

Proof. The initial search tree state has the root node as its current start node. This start node can only be changed by the **Split**, **Move Down** and **Move Up (Internal Node)** transitions. The **Split** transitions change the current start node

to a node that is the end node of a new edge that is added to $\bar{E}_p$. The **Move Down** and **Move Up (Internal Node)** transitions change the start node to a node that is already in $\bar{E}_p$. □

In the following lemma, we use the notation: $N_1 \frown N_2 \frown \ldots \frown N_n$ to denote a sequence of edges $N_1 \frown N_2, N_2 \frown N_3, \ldots, N_{n-1} \frown N_n$ and by stating that $N_1 \frown \ldots \frown N_n \subseteq \bar{E}_p$ we mean that all edges in the sequence are part of $\bar{E}_p$.

Lemma 3. *For every non-root node N in $\bar{E}_p$, there exists a series of edges $N_{root} \frown \ldots \frown N \subseteq \bar{E}_p$, that connect it with the root node N_{root}.*

Proof. We prove by induction of the number of different nodes in $\bar{E}_p$. If all nodes in $\bar{E}_p$ are connected with the root node, then a new node that is added, is also connected to the root node. This is because new nodes can only appear as the end node of an edge $N \frown N'$ that has been added by the **Split** transition. By Lemma 2, the start node N of such an edge is either already in $\bar{E}_p$ or it is the root node. If the start node N is in $\bar{E}_p$ then because of the induction hypothesis, it is connected to the root node by a series of edges $N_{root} \frown \ldots \frown N$ and by extending this series with the edge $N \frown N'$, the new node N' is also connected to the root node. If the start node N is the root, the result trivially holds.

The first edge that is added to $\bar{E}_p$ must have the root node as its start node because of Lemma 2 and so the first new node in $\bar{E}_p$ is also connected to the root node. □

Corollary 1. *For every two nodes N_1 and N_2 in $\bar{E}_p$, there exists a node N_c for which there exists a sequence of edges $N_c \frown \ldots \frown N_1 \subseteq \bar{E}_p$ and a sequence of edges $N_c \frown \ldots \frown N_2 \subseteq \bar{E}_p$.*

Proof. Because of Lemma 3 this holds for $N_c = N_{root}$ (the root node) and it may potentially hold for other nodes as well. □

Lemma 4. *For every edge $N_1 \frown N_2 \in \bar{E}_p$, there exists a derivation $N_1' \rightarrowtail^*_{\omega_r} N_2$ where N_1' is the result of the following extra transition:*

8. Choose $\langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T\rangle_n \rightarrowtail \langle [A_i \mid A], S, B, T\rangle_n$ *for some i, $1 \leq i \leq n$.*

Proof. Every edge $\langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T\rangle_n \frown \langle [A_1' \vee \ldots \vee A_m' \mid A'], S', B', T'\rangle_{n'}$ in $\bar{E}_p$ is created by the **Split** transition and is the result of a derivation $\langle [A_i \mid A], S, B, T\rangle_n \rightarrowtail^*_{\omega_r} \langle [A_1' \vee \ldots \vee A_m' \mid A'], S', B', T'\rangle_{n'}$. □

This result can be easily generalized to a sequence of edges. The following lemma is important for termination:

Lemma 5. *There is no sequence of edges $N \frown \ldots \frown N \subseteq \bar{E}_p$*

Proof. If such a sequence exists, then by Lemma 4, there is a derivation using transitions of ω_r and the extra **Choose** transitions only that transform N into N. The only way that a disjunction can be added to the activation stack is by applying a **Simplify** or **Propagate** transition. These transitions either change the CHR constraint store or add propagation history tuples, so it is not possible to return to the same state again. □

Now we are ready to prove the following result:

Lemma 6. *There exists a terminating derivation $\langle \epsilon, N', [\sigma_i \mid \bar{E}_u], \bar{E}_p\rangle \rightarrowtail^*_{\omega_\lambda} \langle \sigma_i, N, \bar{E}_u, \bar{E}_p\rangle$ with node $N = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T\rangle_n$ and edge $\sigma_i = \langle [A_i \mid A], S, B, T\rangle_n$ and no edge can become active before σ_i does.*

Proof. The only transitions that can be applied to a search tree state with no current CHR execution state (i.e. no active edge), are the **Next**, **Move Down** and **Move Up (Internal Node)** transitions. If the **Next** transition can be applied (i.e. $N' = N$), the theorem trivially holds. Otherwise, because of Lemma 1 and Corollary 1, there exists a node N_c for which there is a sequence of edges $S_1 = N_c \frown \ldots \frown N \subseteq \bar{E}_p$ and a sequence of edges $S_2 = N_c \frown \ldots \frown N' \subseteq \bar{E}_p$. As long as the current start node is not an ancestor node of N, only the **Move Up (Internal Node)** transition is applicable and the current start node becomes 'closer' to the root node. Then as soon as the current start node is an ancestor node of N, only the **Move Down** transition is applicable and remains the only applicable transition until reaching N. Because of Lemma 5, this process terminates. Also, no transition other than **Next** can activate an edge and the list of unprocessed edges $\bar{E}_u$ cannot be changed in the absence of an active edge. □

The following two theorems proof the operational equivalence:

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 $S_1 \rightarrowtail^*_{\omega_\lambda} S_2$ exists such that $\alpha(S_1) = \bar{E}_1$ and $\alpha(S_2) = \bar{E}_2$.*

Proof. We have $\bar{E}_1 = [\sigma \mid \bar{E}]$ because no transitions are possible for an empty list of alternatives. Consider a search tree state S_1 satisfying $\alpha(S_1) = \bar{E}_1$. Such a state has the form $\langle \epsilon, N, \bar{E}_1, \bar{E}_p \rangle$ or the form $\langle \sigma, N, \bar{E}, \bar{E}_p \rangle$. Because of Lemma 6, there exists a derivation $\langle \epsilon, N', \bar{E}_1, \bar{E}_p \rangle \rightarrowtail^*_{\omega_\lambda} \langle \sigma, N, \bar{E}, \bar{E}_p \rangle$, so we can focus on the latter form. Now one of the following cases holds:

- There exists a transition $\sigma \rightarrowtail_{\omega_r} \sigma'$. The only applicable transition in $\omega_r^\vee$ is the **Derive** transition, which transforms $\bar{E}_1$ into $\bar{E}_2 = [\sigma' \mid \bar{E}]$. In ω_λ, the **Derive** transition is also the only applicable transition and it transforms $S_1 = \langle \sigma, N, \bar{E}, \bar{E}_p \rangle$ into $S_2 = \langle \sigma', N, \bar{E}, \bar{E}_p \rangle$ and we have $\alpha(S_2) = \bar{E}_2$.
- $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$. The only applicable transition in $\omega_r^\vee$ is a **Split** transition, which transforms $\bar{E}_1$ into either $\bar{E}_2 = [\sigma_1, \ldots, \sigma_m \mid \bar{E}]$ (depth first) or $\bar{E}_2 = \bar{E} +\!\!+ [\sigma_1, \ldots, \sigma_m]$ (breadth first) with $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$ for $1 \leq i \leq m$. Again in ω_λ, the only applicable transition is also a **Split** transition. It transforms the search tree state $S_1 = \langle \sigma, N, \bar{E}, \bar{E}_p \rangle$ into the state $S_2 = \langle \epsilon, \sigma, [\sigma_1, \ldots, \sigma_m \mid \bar{E}], \bar{E}_p \cup \{N \frown \sigma\} \rangle$ (depth first) or $S_2 = \langle \epsilon, \sigma, \bar{E} +\!\!+ [\sigma_1, \ldots, \sigma_m], \bar{E}_p \cup \{N \frown \sigma\} \rangle$ (breadth first) and $\alpha(S_2) = \bar{E}_2$.
- σ is a final CHR execution state. The only applicable $\omega_r^\vee$ transition is the **Next** transition. This transforms $\bar{E}_1 = [\sigma \mid \bar{E}]$ into $\bar{E}_2 = \bar{E}$. In ω_λ the only applicable transition is either the **Move Up (Successful Leaf Node)** transition (if σ is a successful final execution state) or the **Move Up (Failed Leaf node)** transition (if σ is a failed final execution state). In either case, they transform $S_1 = \langle \sigma, N, \bar{E}, \bar{E}_p \rangle$ into $S_2 = \langle \epsilon, N, \bar{E}, \bar{E}_p \rangle$ and $\alpha(S_2) = \bar{E}_2$.

This concludes our proof. □

Theorem 2. *For every two search tree states S_1 and S_2 for which holds that $S_1 \rightarrowtail_{\omega_\lambda} S_2$, 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 $\alpha(S_2) = \bar{E}_2$.*

Proof. A search tree state S_1 either has the form $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle$ or the form $\langle \epsilon, N, [\sigma \mid \bar{E}_u], \bar{E}_p \rangle$. Since no transitions are applicable if the list of unprocessed edges is empty, we do not need to consider this case (hence σ always exists). For both forms we have that $\alpha(S_1) = [\sigma \mid \bar{E}_u]$. Now we first consider the case that an edge is active (i.e. the current CHR execution state is not ϵ). In this case, one of the following holds:

- There exists a transition $\sigma \rightarrowtail_{\omega_r} \sigma'$. The only applicable transition in ω_λ is the **Derive** transition, which transforms $S_1 = \langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle$ into $S_2 = \langle \sigma', N, \bar{E}_u, \bar{E}_p \rangle$. In $\omega_r^\vee$, the only transition that can be applied on $\bar{E}_1 = [\sigma \mid \bar{E}_u]$ is also the **Derive** transition and the result is $\bar{E}_2 = [\sigma' \mid \bar{E}_u]$ for which holds that $\alpha(S_2) = \bar{E}_2$.
- $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$. The only applicable transition in ω_λ is a **Split** transition, which transforms $S_1 = \langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle$ into either $S_2 = \langle \epsilon, \sigma, [\sigma_1, \ldots \sigma_m \mid \bar{E}_u], \bar{E}_p \cup \{N \frown \sigma\} \rangle$ (depth first) or $S_2 = \langle \epsilon, \sigma, \bar{E}_u +\!\!+ [\sigma_1, \ldots, \sigma_m], \bar{E}_p \cup \{N \frown \sigma\} \rangle$ (breadth first) with $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$ for $1 \leq i \leq m$. In $\omega_r^\vee$, the only transition that can be applied on $\bar{E}_1 = [\sigma \mid \bar{E}_u]$ is also a **Split** transition, which either results in $[\sigma_1, \ldots, \sigma_m \mid \bar{E}_u]$ (depth first) or in $\bar{E}_u +\!\!+ [\sigma_1, \ldots, \sigma_m]$ (breadth first).In either case, it holds that $\alpha(S_2) = \bar{E}_2$.
- σ is a final CHR execution state. If it is a successful final execution state, the only applicable ω_λ transition is **Move Up (Successful Leaf Node)** which transforms $S_1 = \langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle$ into $S_2 = \langle \epsilon, N, \bar{E}_u, \bar{E}_p \rangle$. The same holds for the **Move Up (Failed Leaf Node)** transition for the case that σ is a failed final execution state. In $\omega_r^\vee$, the only applicable transition on $\bar{E}_1 = [\sigma \mid \bar{E}_u]$ if σ is a final CHR execution state, is the **Next** transition, which results in $\bar{E}_2 = \bar{E}_u$ and it holds that $\alpha(S_2) = \bar{E}_2$.

If no edge is currently active (i.e. the current CHR execution state is equal to ϵ), one of the following transitions can be applied in ω_λ: **Next**, **Move Down** or **Move Up (Internal Node)**. For all these transition holds that $\alpha(S_1) = \alpha(S_2)$ if S_2 is the result of applying the transition to S_1. This concludes our proof. □

4.3 State Changes

The **Move Down** transition presented in this Section, can be efficiently implemented if we can store the changes between a given CHR execution state and one that is derived from it, or in other words, if we can create an explicit representation of the trail. We now give an overview of all types of changes that are possible:

- CHR constraints are removed from the CHR constraint store S by the **Simplify** and **Propagate** transitions. We denote these constraints by S_-
- CHR constraints are added to the CHR constraint store by the **Activate** transition. We denote these constraints by S_+.
- Built-in constraints are added to the built-in constraint store by the **Solve** transition and by the **Simplify** and **Propagate** transitions for matching substitutions. We denote these constraints by B_+.
- Propagation history tuples are added to the propagation history by the **Propagate** transition. We denote these tuples by T_+.
- The next free identifier number is raised by the **Activate** transition. We denote the raise in value by n_+.
- The activation stack is changed in every transition. Although a more detailed description of the exact changes can be given, it is sufficient to say that a transition changes a given activation stack A into another activation stack A'.

So we conclude that every derivation can be written as $\langle A, S \cup S_-, B, T \rangle_n \rightarrowtail^*_{\omega_r} \langle A', S \cup S_+, B \wedge B_+, T \cup T_+ \rangle_{n+n_+}$.

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 [14] 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 Global Variables in SWI-Prolog

The global variables facility of SWI-Prolog allows creating associations between an atom (key) and a term (value). It supports two ways of creating such an association: using backtrackable assignment and using non-backtrackable assignment. The effect of using backtrackable assignment (i.e. the association that is generated) is undone when backtracking to a point before the call. By using non-backtrackable assignment, the association remains. An important issue here is that the creation of the term representing the value of the global variable, may also be undone by backtracking.

One way to deal with this is by storing a copy of the term instead of storing the original term itself. This way, the original term can be safely deconstructed without influencing the copy. Alternatively, the link is kept and the stored term is deconstructed as well. SWI-Prolog offers both alternatives, the former by using the `nb_setval/2` predicate, the latter by using the `nb_linkval/2` predicate.

We use non-backtrackable assignment intensively, amongst others to record how a derivation changed the CHR execution state and to implement the search order. We use a combination of `nb_setval/2` and `nb_linkval/2` to store variable bindings.

5.2 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. This is done as follows: if the current start node 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 (if it is not already processed, the next edge's start node cannot exist). 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.

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 the start node of that edge. 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 combination of `next_edge/1` and `same_branch/2` must be tried again when the value of `next_edge/1` changes (when an edge of the list of unprocessed edges has been processed). How this is implemented is not important. If the call to `same_branch/2` fails, the changes made by the current active edge are undone.

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

```
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. Then one of the following three events can happen:

- The derivation ends in a failed final execution state. The CHR execution state changes of the edge are backtracked automatically.
- The derivation ends in a successful final execution state. A solution has been found. An external event (e.g. from the toplevel) may ask for further solutions.
- The derivation ends with a **Split** transition. First, the current active edge is marked as processed and the CHR execution state changes that it created are collected and stored. Then, for every alternative choice of the choice point, a new unprocessed edge is created. Finally, the next unprocessed edge is activated.

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.

Example 1. A small example illustrates how we can move from one node of the search tree to another. In Figure 2 we see a search tree consisting of 7 nodes and 6 edges. The nodes are numbered in the order in which they are visited (i.e. breadth first) and the root node is labeled as R. Consider that we have just processed the edge $2 \frown 5$, so the current start node is node 5. The next unprocessed edge is the edge $3 \frown 6$.[2] The first call to `activate_next_edge/1` fails because there is no branch that contains both $2 \frown 5$ and $3 \frown 6$. We backtrack to the most recent choice point: the `activate_next_edge/1` at node 2 (i.e. at the end of processing edge $1 \frown 2$). Again the call to `activate_next_edge/1` fails and we backtrack to the `activate_next_edge/1` call at node 1. This call succeeds because the edge $R \frown 1$ and the edge $3 \frown 6$ are part of the same branch. Since the next (processed) edge on the path between node 1 and node 3 is the edge $1 \frown 3$, we activate this edge. This brings us to node 3. Finally, the call to `activate_next_edge/1` in this node, causes the unprocessed edge $3 \frown 6$ to become active, removing it from the list of unprocessed edges.

[2] In fact node 6 has not been discovered yet and might not even exist, but it makes it easier to refer to the edge.

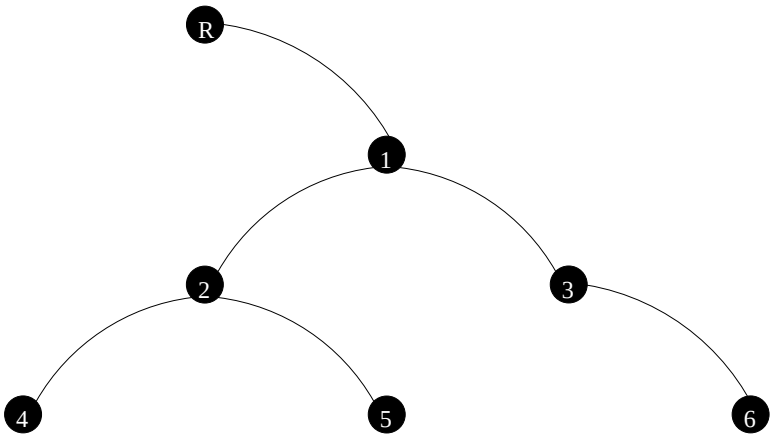

Figure2. Search Tree

5.3 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). Because of what we described in Section 4.3, the derivation of an edge has the form $\langle [A_i \mid A], S \cup S_-, B, T\rangle_n \rightarrowtail^*_{\omega_r} \langle [A'_1 \vee \ldots \vee A'_m \mid A'], S \cup S_+, B \wedge B_+, T \cup T_+\rangle_{n+n_+}$, 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 set of added CHR constraints S_+, the set of removed CHR constraints S_-, the set T_+ of added propagation history tuples, the conjunction B_+ of added built-in constraints and the integer n_+ that denotes the increase of the next free identifier.

Our approach is similar to the explicit trailing mechanism of [15], 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_+. We present a program transformation that adds explicit identifiers to the constraints and makes the propagation history explicit.

For every CHR constraint `c/n`, we introduce an *id-extended* CHR constraint `c_id/n+1` and add a rule of the form:

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

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

and every propagation rule of the form

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

is converted into a rule

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

The pragma `no_history` turns the implicit propagation history off. This is not necessary for the correctness of our approach, but it makes the proof easier and can cause some speedup. An implementation for the `in_history/1` and `add_to_history/1` predicates is described further on.

To prove that our program transformation does not change the operational behavior of a CHR program, we first define a mapping function f_{map} as follows:

Definition 2. *The mapping function f_{map} maps a CHR execution state E of the original program P on a tuple $\langle E', T' \rangle$ where E' is an execution state and T' an explicit propagation history, of the transformed program P'. This mapping function is defined as follows:*

- $f_{map}(\langle A, S, B, T \rangle_n) = \langle \langle A', S', B, \emptyset \rangle_{(2 \cdot n-1)}, T \rangle$
- *We get A' from A by replacing every identified CHR constraint $c(\bar{X})\#i$ and every occurrenced identified CHR constraint $c(\bar{X})\#i : j$ with respectively the id-extended identified CHR constraint $c_{id}(i, \bar{X})\#2 \cdot i$ and the id-extended occurrenced CHR constraint $c_{id}(i, \bar{X})\#2 \cdot i : j$.*
- *We get S' from S by replacing every identified CHR constraint $c(\bar{X})\#i$ with the id-extended identified CHR constraint $c_{id}(i, \bar{X})\#2 \cdot i$.*

The following theorem uses f_{map} to prove the correctness of our approach.

Theorem 3. *For every transition $E \rightarrowtail_{\omega_r} E'$ in the original program P, there exists a unique derivation $f_{map}(E) \rightarrowtail^*_{\omega_r} f_{map}(E')$ in the transformed program P'.*

Proof. To prove our theorem, we look at each transition in detail. We make use of equality propagation of [1] intensively to eliminate built-in constraints.

1. Solve For $E = \langle [c \mid A], S_0 \cup S_1, B, T \rangle_n$, $f_{map}(E) = \langle \langle [c \mid A'], S'_0 \cup S'_1, B, \emptyset \rangle_{2 \cdot n-1}, T \rangle$. Since the built-in constraint c is not affected by the mapping function, the only applicable transition in P' is also the **Solve** transition. We now prove that this transition preserves the mapping.

Consider that S'_0 consists of the id-extended versions of the constraints in S_0 and S'_1 consists of the id-extended versions of the constraints in S_1, then we have $vars(S'_0) = vars(S_0)$ and $vars(S'_1) = vars(S_1)$. Because $vars(S_0) \subseteq fixed(B)$, also $vars(S'_0) \subseteq fixed(B)$ and similarly, because $vars(S_1) \not\subseteq fixed(B)$, also $vars(S'_1) \not\subseteq fixed(B)$.
This means that application of **Solve** to $\langle [c \mid A'], S'_0 \cup S'_1, B, \emptyset\rangle_{2\cdot n-1}$ leads to $\langle [S'_1 \mid A'], S'_0 \cup S'_1, B, \emptyset\rangle_{2\cdot n-1}$. The propagation history is not changed. We conclude that a **Solve** transition transforms a state E into a state E' if and only if a **Solve** transition transforms a tuple $f_{map}(E)$ into a tuple $f_{map}(E')$.

2. **Activate** For $E = \langle [c \mid A], S, B, T\rangle_n$, $f_{map}(E) = \langle\langle [c \mid A'], S', B, \emptyset\rangle_{2\cdot n-1}, T\rangle$. The new CHR constraint c is not affected by the mapping, because it is not an identified or occurrenced identified constraint. This means that the only applicable transition on $f_{map}(E)$ in P' is the **Activate** transition. Application of this transition leads to $\langle [c\#2\cdot n-1:1 \mid A'], \{c\#2\cdot n-1\} \cup S', B, \emptyset\rangle_{2\cdot n}$.
The first and only occurrence of the constraint c is in the rule $c(\bar{X}) \iff new_id(ID), c_{id}(ID, \bar{X})$. Since this rule has no guard and we can trivially create a matching substitution θ, we can apply the **Simplify** transition which, after applying equality propagation for the matching substitution, results in $\langle [new_id(ID), c_{id}(ID, \bar{X}) \mid A'], S', B, \emptyset\rangle_{2\cdot n}$.
$new_id(ID)$ is a built-in constraint that binds the variable ID to the value of the next free explicit identifier: n. A **Solve** transition, followed by equality propagation on the constraint $ID = n$, results in $\langle [c_{id}(n, \bar{X}) \mid A'], S', B, \emptyset\rangle_{2\cdot n}$. Because the built-in constraint $ID = n$ does not affect any constraint in S', there is no need to reconsider constraints in S'.[3]
The next constraint on the execution stack is a new CHR constraint which causes an **Activate** transition. This transition results in $\langle [c_{id}(n, \bar{X})\#2\cdot n:1 \mid A'], \{c\#2\cdot n\} \cup S', B, \emptyset\rangle_{2\cdot(n+1)-1}$. The propagation history is not changed.
This proves that the **Activate** transition is preserved by our mapping.

3. **Reactivate** $f_{map}(\langle [c\#i \mid A], S, B, T\rangle_n) = \langle\langle [c_{id}\#2\cdot i \mid A'], S', B, \emptyset\rangle_{2\cdot n-1}, T\rangle$ so in P', the **Reactivate** transition is also the only possible transition as the first element of the execution stack is an identified CHR constraint. This transition trivially preserves our mapping.

4. **Drop** $f_{map}(\langle [c\#i : j \mid A], S, B, T\rangle_n) = \langle\langle [c_{id}\#2\cdot i : j \mid A'], S', B, \emptyset\rangle_{2\cdot n-1}, T\rangle$. Since the id-extended constraint c_{id} has the same number of occurrences in the transformed program as c in the original program, the precondition of **Drop** holds in the transformed program and it is the only transition that can fire. So also this transition trivially preserves our mapping.

5. **Simplify** For $E = \langle [c\#i : j \mid A], \{c\#i\} \cup H_1 \cup H_2 \cup H_3 \cup S, B, T\rangle_n$, $f_{map}(E) = \langle\langle [c_{id}\#2\cdot i : j \mid A'], \{c_{id}\#2\cdot i\} \cup H'_1 \cup H'_2 \cup H'_3 \cup S', B, \emptyset\rangle_{2\cdot n-1}, T\rangle$. The rule r in the original program is transformed into an equivalent rule where all original constraints are replaced by id-extended constraints. The constraints in H'_1, H'_2 and H'_3 are id-extended versions of the constraints in respectively H_1, H_2 and H_3. We can create a matching substitution $\theta' = \theta \wedge \theta_{id}$ with θ the corresponding matching substitution in the original program and θ_{id} a trivial matching substitution for the explicit identifiers.
Because the variables in θ_{id} are all fresh variables, and because it trivially holds that $\mathcal{D} \models \bar{\exists}_B \theta_{id}$, we have $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g) \iff \mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge \theta_{id} \wedge g)$ which means that the **Simplify** transition is applicable on a state E in the original program P, if and only if it is applicable on $f_{map}(E)$ in the transformed program P'.

[3] In the refined operational semantics, all constraints that contain variables that are not fixed by the built-in store, will be reconsidered. Practical implementations only reconsider a subset of these constraints. In a 'good' program, this should only be a matter of efficiency and not of correctness.

Application of the **Simplify** transition results, after applying equality propagation for the matching substitution θ_{id}, in $\langle C +\!\!+ A', H_1' \cup S', B \wedge \theta \wedge g, \emptyset\rangle_{2\cdot n-1}$. Since the body C does not contain identified CHR constraints, it is not affected by our mapping and can be added safely to the execution stack. This proves that the **Simplify** transition is preserved by our mapping.

6. **Propagate** For $E = \langle [c\#i : j \mid A], \{c\#i\} \cup H_1 \cup H_2 \cup H_3 \cup S, B, T\rangle_n$, $f_{map}(E) = \langle\langle [c_{id}\#2\cdot i : j \mid A'], \{c_{id}\#2\cdot i\} \cup H_1' \cup H_2' \cup H_3' \cup S', B, \emptyset\rangle_{2\cdot n-1}, T\rangle$. The case of the **Propagate** transition is similar to that of the **Simplify** transition. The most important difference is that a propagation history is used for pure propagation rules (rules for which H_3 is empty).
   In the original program, it is checked whether a tuple $t = id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ [r]$ is in the propagation history T. In the transformed program, this test is moved to the guard. The guard of the transformed program is $g' = not(in_history(id(H_1') +\!\!+ [i] +\!\!+ id(H_2') +\!\!+ [r])) \wedge g$ where the id function applied to an id-extended CHR constraint returns the explicit identifier of the constraint. Identifiers for the rules are generated at compile time.
   The `in_history/1` predicate checks whether the tuple that is its argument, is already in the propagation history. The negation of it has the semantics of negation as finite failure. Therefore, we cannot consider it as a built-in constraint. Also, since it is not allowed to add CHR constraints to the store in the guard, we cannot consider it as a CHR constraint. The `add_to_history/1` predicate adds the tuple that is its argument, to the explicit propagation history.
   So we have that a **Propagate** transition in the original program P can be applied on a state E if and only if a **Propagate** transition can be applied on $f_{map}(E)$ in the transformed program P'. After application, the implicit history of E and the explicit history of $f_{map}(E)$ are the same.
7. **Default** $f_{map}(\langle [c\#i : j \mid A], S, B, T\rangle_n) = \langle\langle [c_{id}\#2 \cdot i : j \mid A'], S', B, \emptyset\rangle_{2\cdot n-1}, T\rangle$. In P, this transition can be applied only if no other transition could be applied for the active constraint $c\#i$ and for occurrence j. This means that an occurrence j exists (otherwise the **Drop** transition was applied) and that the **Simplify** or **Propagate** transition for the given occurrence could not be applied. For the latter two transitions, we have shown that their applicability is not changed by the mapping: a **Simplify** or **Propagate** transition can be applied on a state E in P, if and only if it can be applied on $f_{map}(E)$ in P'. This means that by elimination, this also holds for the **Default** transition. So also this transition is trivially preserved by our mapping.

This proves the induction step and ends our proof. □

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: the already mentioned *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`. The rule to add an explicit identifier is extended to create a new element in S_+ as follows:

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

The transformation of simplification and propagation rules is extended as follows:

```
r @ normal, c_id_1(ID_1,X̄_1), ..., c_id_i(ID_i,X̄_i) \
    c_id_{i+1}(ID_{i+1},X̄_{i+1}), ..., c_id_n(ID_n,X̄_n) <=>
        guard |
        c_del_n_{i+1}(ID_{i+1}),
        ...,
        c_del_n_n(ID_n),
        prelink(t(X̄_1,...,X̄_n)),
        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 the store and removed from it on the same 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. We can load the CHR constraint store changes by first calling the constraints in the saved list, which puts them into the CHR constraint store, and then applying the following rules for every constraint `c/n`:

```
perform_changes \ c_add(ID,X̄) <=>
        c_id(ID,X̄).
perform_changes \ c_del_n(ID), c_id(ID,X̄) <=>
        true.
```

Changes to the Propagation History The transformation of propagation rules is extended as follows:

```
r @ normal, c_id_1(ID_1,X̄_1), ..., c_id_n(ID_n,X̄_n) ==>
        \+ in_history(t(ID_1, ..., ID_n, r)),
        guard |
        add_to_history(t(ID_1,...,ID_n,r)),
        prelink(t(X̄_1,...,X̄_n)),
        body.
```

Here we have added the already described `normal` constraint. We also added a call to `prelink/1` like for simplification and simpagation rules. Propagation rules do not remove constraints, so we do not need to add elements to S_-.

The explicit propagation history can be implemented as a list of tuples, for example by using the global variables of SWI-Prolog:

```
in_history(Tuple) :-
        b_getval(history,List),
        member(Tuple,List).

add_to_history(Tuple) :-
        b_getval(history,List),
        b_setval(history,[Tuple|List]).
```

By comparing the length of the history lists at the start node and at the end node of the active edge, it is trivial to find the tuples that have been added in the derivation.

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.[4] 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 variables that occur in the initial goal can be stored in the list of variables right before calling the goal. Variables that already occurred in the CHR constraint store, are not in the initial goal, and are potentially bound while processing the active edge, must come from one of the constraints in the head of a rule. We can add them to the list of variables in the body of the rule. Finally, variables that were not in the CHR constraint store before processing the edge, can be added to the list of variables in the body of the rule that transforms a constraint into an id-extended constraint.

The `prelink/1` predicate that we used in the transformations of the original program rules, adds the new variables in the term that is its argument, to the original list of variables.

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 benchmarks are performed on a 2.8 GHz Pentium IV processor using SWI-Prolog version 5.6.0. They form 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.

For finding all solutions of a Sudoku puzzle with 295 different solutions, we get the following times:

Search Strategy	time(s)
Breadth First	50.14
Depth First (Explicit)	6.12
Depth First (Implicit)	2.04

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

and for finding all solutions of the n queens problem for different n, we get the following times (in seconds):

n	Breadth First	Depth First (Explicit)	(Implicit)
7	1.78	1.16	0.27
8	9.97	5.54	1.38
9	66.22	28.27	8.73

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 the Sudoku benchmark, these hash tables are responsible for at least a third of the runtime.

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 [18] 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 if the host language supports the specification of different search strategies, we can extend this without much effort to CHR implementations that are compiled into this host language, like CHR for 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 [17]. 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 [16] copying is combined with recomputation and compared with trailing. The Java Abstract Search Engine (JASE) [13] 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,[5] 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 [14] and SWI-Prolog [20] only. This implementation introduces an explicit trailing mechanism that is an extension of the one in [15]. 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 Constraint Logic Programming 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 *FQAS '98: Proceedings of the Third International Conference on Flexible Query Answering Systems*, pages 1–14, London, UK, 1998. Springer-Verlag.
4. Maurice Bruynooghe. Enhancing a search algorithm to perform intelligent backtracking. *TPLP*, 4(3):371–380, 2004.

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

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. INCLP(R) - Interval-based nonlinear constraint logic programming over the reals. In Fink et al. [7], pages 91–100.
13. Ekkehard Krämer. A generic search engine for a Java constraint kit. Master's thesis, Institute of Computer Science, LMU, Munich, Germany, January 2001.
14. 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.
15. 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.
16. Christian Schulte. Comparing trailing and copying for constraint programming. In *ICLP*, pages 275–289, 1999.
17. 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.
18. Terrance Swift and David Scott Warren. An abstract machine for SLG resulution: Definite programs. In *SLP*, pages 633–652, 1994.
19. Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Tom Schrijvers and Thom Früwirth, editors, *Proceedings of the 2nd Workshop on Constraint Handling Rules*, pages 47–62, October 2005.
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.