CHR Exhaustive Execution - Revisited

AHMED ELSAWY[1], AMIRA ZAKI[1,2], SLIM ABDENNADHER[1]
[1] *German University in Cairo, Egypt;* [2] *Ulm University, Germany*
(*e-mail:* {ahmed.el-sawy,amira.zaki,slim.abdennadher}@guc.edu.eg)

submitted 29 April 2015; accepted 5 June 2015

Abstract

Constraint Handling Rules (CHR) apply guarded rules to rewrite constraints in a constraint store, until a final state is reached in which no more rules are applicable. The rules cannot be retracted, therefore CHR does not backtrack over alternatives. In this paper, a novel source-to-source transformation is proposed, which transforms any given CHR program to one that backtracks over all possible alternatives. Accordingly, execution of the transformed program finds all possible results that are entailed from the complete search tree of the source program. Compared to a previous approach presented, the newly introduced transformation generates a search tree without redundant computations. Furthermore, the approach is generalized for all types of CHR rules.

KEYWORDS: Declarative programming, Constraint Handling Rules, Exhaustive execution, Source-to-source transformation, Full-search space exploration

1 Introduction

Constraint Handling Rules (CHR) is a rule-based programming language, which was initially designed for implementing constraint solvers but has proven to become an elegant general-purpose language with a large spectrum of applications (Frühwirth 2009). Rules rewrite a multi-set of constraints until a final state is reached where no more rules are applicable (Frühwirth et al. 1996). The execution is committed-choice and fired rules cannot be retracted, thus CHR can not backtrack over alternatives.

The execution of a query by a CHR program is known as a derivation from an initial state to a final one. The derivation path depends on the implementation of the CHR system, which invokes a deterministic handling for various factors. These include the order of constraints within the query, the order of rules within the program and the order of constraints within the rules.

For confluent programs, this is not a problem because all derivation paths ultimately lead to the same final state. However non-confluent programs, such as those used for agent programming, would produce different final states depending on the chosen derivation path. Due to the committed-choice nature of CHR, it might not be possible to reach all final states for these programs. Hence the need arises to implement a mechanism to enforce search space exploration of a CHR derivation.

The rule-based agent planning Blocks World example (Duck et al. 2007; Lam and Sulzmann 2006) can be modeled in CHR to describe the behavior of an agent in a world of objects (Elsawy et al. 2014). An agent holding an object `X` can be represented by a `hold(X)` constraint. The agent picks-up an object `X` in a `get(X)` constraint. An agent not holding any objects can be represented by an `empty` constraint, and an object `Y` that has been put-down can be represented with a `clear(Y)` constraint. The first CHR rule below allows an empty-handed agent who should get an object `X` to hold it. The second rule allows an agent that is holding an object `Y` and wants to get another object `X` to place down `Y` as a clear object and then hold `X`, the agent has to clear object `X`, because it can carry at most one object.

Example 1 (**Blocks World)**

```
rule-1 @ get(X), empty <=> hold(X).
rule-2 @ get(X), hold(Y) <=> hold(X), clear(Y).
```

An empty-handed agent who should pick-up a box and a cup is expressed with an initial query '`empty, get(box), get(cup)`'. There are two possible scenarios from this state; the first path would have the agent get the box first then the cup, hence ending up to be holding the cup. This final state would be expressed by '`hold(cup), clear(box)`'. The other scenario is that the agent gets the cup first, then clears it and ends up holding the box; which is expressed as '`hold(box), clear(cup)`'. These two scenarios can be depicted in Figure 1 as two disjoint final states. The CHR implementation of the K.U. Leuven system (Frühwirth and Raiser 2011) entails that only one final state is reached '`hold(cup), clear(box)`'.

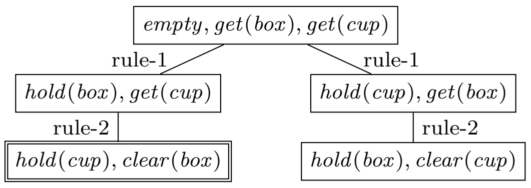


Fig. 1: Derivation tree for Example 1 with query: `empty,get(box),get(cup)`, only the left final state with a double border is reached using the K.U. Leuven system

CHR runs on top of a host language like Prolog which provides evaluation of built-in constraints. Constraint Handling Rules with Disjunction (CHR$^\vee$) is an extension of CHR (Abdennadher and Schütz 1998) which allows disjunctive bodies and hence introduces Prolog's backtracking over alternatives. Application of a transition rule generates a branching derivation which can be utilized to overcome the committed-choice execution of CHR.

Source-to-source transformations can be used to transform CHR programs into extended ones with additional machinery. This work aims to transform a CHR committed-choice derivation to a traversable search-tree; the idea was introduced in (Elsawy et al. 2014) as a source-to-source transformation. This involved transforming the CHR program into an extended CHR program featuring exhaustive completion to fully explore a goal's search space.

The transformation of (Elsawy et al. 2014) involved changing a derivation into a search tree with disjunctive branches, such that it can be exhaustively traversed to reach all leaves. The approach involved annotating rule constraints with their occurrence within the program and the current tree depth while searching. An initial query was transformed into one that would trigger all different permutations of the two reference arguments, and hence this would generate a complete tree of all possible constraint/rule matchings. For the Blocks World example, part of the transformed derivation tree in shown in Figure 2.

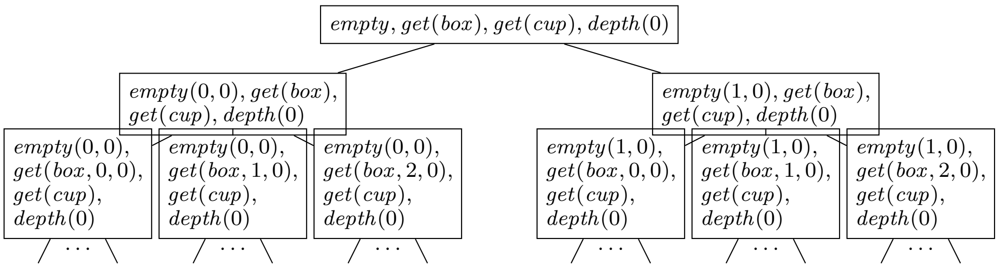

empty, get(box), get(cup), depth(0)
empty(0,0), get(box), get(cup), depth(0)
empty(1,0), get(box), get(cup), depth(0)
empty(0,0), get(box,0,0), get(cup), depth(0)
empty(0,0), get(box,1,0), get(cup), depth(0)
empty(0,0), get(box,2,0), get(cup), depth(0)
empty(1,0), get(box,0,0), get(cup), depth(0)
empty(1,0), get(box,1,0), get(cup), depth(0)
empty(1,0), get(box,2,0), get(cup), depth(0)

Fig. 2: Part of derivation tree of same query but under transformed exhaustive program of (Elsawy et al. 2014). Annotations are ignored for equivalence of constraints, i.e. `get(box,_,_)` is equivalent to `get(box)`, etc.

However, there were two main problems with the transformation. First, the search tree generated contained several duplicated branches. The tree was complete, however many redundant matchings were tried, which could be optimized greatly. Secondly, the approach was presented for only one type of CHR rules known as simplification rules. The transformation requires an additional handling for a propagation history to handle other CHR rule types (known as propagation and simpagation rules), yet the details of this was not formally investigated.

Alternatively, the angelic semantics of CHR (Martinez 2011) is a similar work aimed to explore all possible execution choices using derivation nets. However, no implementation nor definition of an operational semantics was provided.

Therefore this work aims to revisit the exhaustive execution problem, by designing a more generic and efficient source-to-source transformation that enforces a full-space exploration of a CHR derivation tree. The new exhaustive transformation will be formalized in this work by defining how it handles all types of CHR rules. An evaluation showing the gained search-space improvement will be also shown in comparison to previous work (Elsawy et al. 2014).

This work also proposes interesting applications for exhaustive CHR, bringing it closer to its declarative form. Exhaustive CHR execution can be utilized to solve complex problems by writing simpler code. For example to find all paths from source to destination in a graph, a simple CHR program can be written to find a single path from source to destination. Then the exhaustive program would automatically find all paths by fully exploring all derivations, without having to write a new program that explicitly explores all paths.

The paper proceeds by introducing Constraint Handling Rules in Section 2. The transformation is presented in Section 3, with evaluation in comparison to the previous approach. Section 4 introduces an application for the exhaustive transformation, then the paper concludes with mention of possible future work in Section 5.

2 Constraint Handling Rules, by example

Constraint Handling Rules (CHR) (Frühwirth 2009; Frühwirth and Raiser 2011) consist of guarded rewrite rules that perform conditional transformation of multi-sets of constraints, known as the constraint store. The rules used in the Blocks World example are simplification rules, which replaces the left-hand side constraints with the right-hand side ones. However, there are two more types of rules, propagation and simpagation rules. All rules have an optional unique identifier separated from the rule body by @. For all rules, H_k and/or H_r form the rule head. H_k and H_r are sequences of user-defined CHR constraints that are kept and removed from the constraint store respectively. *Guard* is the rule guard consisting of a sequence of built-in constraints, and B is the rule body comprising of CHR and built-in constraints. The generalized rules are of the forms shown below:

$$\begin{gathered} \textit{simpagation-id} \texttt{ @ } H_k \setminus H_r \Leftrightarrow \textit{Guard} \mid B \\ \textit{simplification-id} \texttt{ @ } H_r \Leftrightarrow \textit{Guard} \mid B \\ \textit{propagation-id} \texttt{ @ } H_k \Rightarrow \textit{Guard} \mid B \end{gathered}$$

An example CHR program with the three types of rules with unique identifiers is shown below; all the rules have true guard expressions which are thus eliminated.

```
simplification @ a, b <=> c.
propagation @ a, b ==> c.
simpagation @ a \ b <=> c.
```

The first simplification rule checks if constraints `a` and `b` are in the store, it then removes them and adds constraint `c`. The second propagation rule adds constraint `c` to the store if constraints `a` and `b` are in the store. The third simpagation rule combines the functionality of the previous two rules. If constraints `a` and `b` are in the store, then the constraints after \ are removed while keeping the ones before it. Here this means removing constraint `b`, keeping constraint `a` and adding constraint `c`. A query `‘a,b’` can have multiple derivation paths, depending on which rule is chosen. All possible derivations are shown in Figure 3.

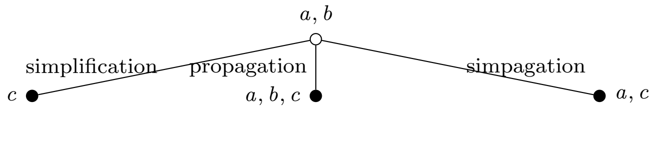


Fig. 3: Complete derivation tree for query `‘a,b’`

(Koninck et al. 2008) depicted a CHR derivation as a tree, consisting of a set of nodes and directed edges connecting these nodes. The root node corresponds to the initial state or goal. A leaf node represents a successful or failed final state. Edges between nodes denote the applied rules. Due to the clarity offered by derivation trees for such derivations, in this work all examples will be depicted using trees.

3 Source-to-Source Transformation

In this work, a transition is defined as a tuple $(\textit{ruleid}, \textit{ID}_1, \ldots, \textit{ID}_m)$, where *ruleid* is a rule identifier from the given program, and $\textit{ID}_1$ to $\textit{ID}_m$ are the identifiers of the constraints in the goal that matched the head constraints of *ruleid*. The transformation is based on the fact that CHR exhaustively applies transitions until a final state is reached where no more transitions are applicable. If transitions $t_1, \ldots, t_n$ are applicable on CHR state G, then CHR will deterministically apply only one of these transitions without retracting. The transformation ensures completeness by deriving two CHR states from state G using disjunction and backtracking of Prolog, when transition t_i for $1 \leq i \leq n$ is applied on G.

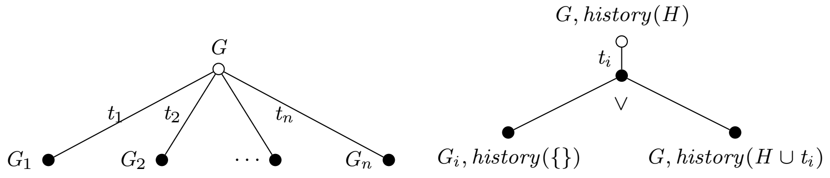


Fig. 4: Left - original program derivation tree, Right -transformed program derivation tree

The first derivation applies transition t_i on state G leading to state G_i, on the other hand, the second derivation makes no changes on state G, except that the transition information is stored in a transition application history. The idea of using a history or token store was used in (Abdennadher 1997) to avoid non-termination due to propagated constraints. Similarly, in this work the transition application history is used to prevent applying the stored transitions a second time before a different transition is applied. Thus CHR will apply an applicable transition t_j on the derived state, if t_j is not in the transition application history. This is shown by Figure 4, the tree on the left represents the application of transitions $t_1, \ldots, t_n$ on state G leading to states $G_1, \ldots, G_n$. The tree on the right shows how the transformed program applies transition t_i and uses disjunction to derive two states representing G_i and G, where the latter state does not allow the application of transition t_i using the transition application history. This will eventually lead to the application of all the remaining transitions.

3.1 Transformation

The transformation extends every user defined constraint by a unique identifier, to be used by the transition application history. Constraint $id/1$ is added to the initial query, initially with the value one. This constraint stores the next unique identifier.

For every head constraint c/n that appears in the source program, the following rule is added to the transformed program:

$$extend\_c \ \texttt{@} \ c(X_1, \ldots, X_n), id(V) \Leftrightarrow c^t(X_1, \ldots, X_n, V), id(V+1)$$

Moreover, our source-to-source transformation applies a specific transformation for every rule type *simplification*, *propagation*, and *simpagation* in the source program.

1. Transformation of *simplification rules* is defined as follows:

Every *simplification* rule in the source program, defined as *(ruleid* @ $H_r \Leftrightarrow$ *Guard* | B*)* is transformed to rule :

$$\begin{array}{c} \mathit{ruleid\_t} \text{ @ } H_r^t, \mathit{history}(L) \Leftrightarrow \mathit{transition} \notin L, \mathit{Guard} \mid \\ B, \mathit{history}(\{\}) \vee H_r^t, \mathit{history}(\mathit{transition} \cup L) \end{array}$$

where $\mathit{transition} = (\mathit{ruleid}, ID_1, \ldots, ID_m)$
$H_r = c_1(X_{11}, \ldots, X_{1n_1}), \ldots, c_m(X_{m1}, \ldots, X_{mn_m})$
$H_r^t = c_1^t(X_{11}, \ldots, X_{1n_1}, ID_1), \ldots, c_m^t(X_{m1}, \ldots, X_{mn_m}, ID_m)$

Transformed simplification rules extend head constraints with the constraint identifier and add the transition application history constraint *history*/1 to the rules' head. These rules are applied if the *Guard* holds and the transition application history does not have a record of applying the rule with the identifiers $ID_1, \ldots, ID_m$. Applying the rule entails two derivations, the first derivation applies the corresponding rule from the source program, it adds the body B of the original rule to the goal, moreover it empties the transition application history. The second derivation adds the removed head constraints H_r^t to the goal and it adds the transition information to the transition application history.

2. A *propagation rule* with rule head H_k would be transformed to a *simpagation rule* with rule head $H_k \setminus \mathit{history}(L)$, since H_k should be kept in the goal and the transition application history should be updated after the rule is applied. However, this approach might lead to a trivial non-termination because the rule could be applied on the same constraints infinitely many times. Moreover, it is inconsistent with the semantics of CHR because a CHR *propagation rule* should not be applied with the same constraints more than once. Thus, for *propagation rules* in the source program, our transformation uses a propagation history *prop_history*/1 besides the transition application history. The propagation history stores the identifiers of the constraints that fired the transformed rule to prevent these constraints from firing the same rule again. In other words, this propagation history simulates the propagation history used by CHR. Transformation of *propagation rules* is defined by the following rule transformation:

Every *propagation* rule in the source program, defined as *(ruleid* @ $H_k \Rightarrow$ *Guard* | *B)* is transformed to rule :

$$\begin{array}{c} \mathit{ruleid\_t} \text{ @ } H_k^t \setminus \mathit{history}(L), \mathit{prop\_history}(P) \Leftrightarrow \mathit{transition} \notin L, \mathit{transition} \notin P, \\ \mathit{Guard} \mid B, \mathit{history}(\{\}), \mathit{prop\_history}(\mathit{transition} \cup P) \vee \\ \mathit{history}(\mathit{transition} \cup L), \mathit{prop\_history}(P) \end{array}$$

where $\mathit{transition} = (\mathit{ruleid}, ID_1, \ldots, ID_m)$
$H_k = c_1(X_{11}, \ldots, X_{1n_1}), \ldots, c_m(X_{m1}, \ldots, X_{mn_m})$
$H_k^t = c_1^t(X_{11}, \ldots, X_{1n_1}, ID_1), \ldots, c_m^t(X_{m1}, \ldots, X_{mn_m}, ID_m)$

The proposed transformation transforms *propagation rules* to *simpagation rules* in which the kept head of the *simpagation rule* is the head of the *propagation rule* and the removed head of the *simpagation rule* is the transition application history and the propagation history. The transformed rule uses the propagation history to guarantee that the kept head constraints do not apply the same rule more than once. Applying the transformed rules entails two derivations, the first derivation adds the body B of the original rule to the goal, empties the transition application history, and stores the transition information to the propagation history. The second derivation does not apply the original rule's body, thus the propagation history remains unchanged and the transition information is added to the transition application history.

3. *Simpagation rules* of the form $(H_k \setminus H_r \Leftrightarrow \mathit{Guard} \mid B)$ are equivalent to *simplification rules* of the form $(H_k, H_r \Leftrightarrow \mathit{Guard} \mid B, H_k)$, therefore our proposed transformation transforms *simpagation rules* from the source programs to *simplification rules*, adds the identifiers to the user defined constraints in the head and the body of the transformed constraints, and applies the *simplification rules* transformation. Thus *simpagation rules* from the source program will be transformed according to the following rule transformation :

Every *simpagation* rule in the source program, defined as $(\mathit{ruleid}\ \texttt{@}\ H_k \setminus H_r \Leftrightarrow \mathit{Guard} \mid B)$ *is transformed to rule :*

$$\mathit{ruleid\_t}\ \texttt{@}\ H_k^t, H_r^t, \mathit{history}(L) \Leftrightarrow \ \mathit{transition} \notin L, \mathit{Guard} \mid$$
$$B, H_k^t, \mathit{history}(\{\}) \vee H_k^t, H_r^t, \mathit{history}(\mathit{transition} \cup L)$$

where $\mathit{transition} = (\mathit{ruleid}, \mathit{ID}_1, \ldots, \mathit{ID}_i, \mathit{ID}_j, \ldots, \mathit{ID}_m)$
$H_k = c_1(X_{11}, \ldots, X_{1n_1}), \ldots, c_i(X_{i1}, \ldots, X_{in_i})$
$H_r = c_j(X_{j1}, \ldots, X_{jn_j}), \ldots, c_m(X_{m1}, \ldots, X_{mn_m})$
$H_k^t = c_1^t(X_{11}, \ldots, X_{1n_1}, \mathit{ID}_1), \ldots, c_i^t(X_{i1}, \ldots, X_{in_i}, \mathit{ID}_i)$
$H_r^t = c_j^t(X_{j1}, \ldots, X_{jn_j}, \mathit{ID}_j), \ldots, c_m^t(X_{m1}, \ldots, X_{mn_m}, \mathit{ID}_m)$

Transformed simpagation rules are applied if the kept and removed head constraints did not previously fire the rule, and if the Guard holds. Firing the rule entails two derivations, the first derivation applies the original rule's body, adds the kept head constraints to the goal, and empties the transition application history, while the second derivation adds the kept head and removed head constraints to the goal since the original rule from the source program will not be applied and it adds the transition information to the transition application history.

3.2 Example

In the following example we apply the transformation on the CHR program of Example 1. We show the transformed program, in addition to the full search tree generated by it.

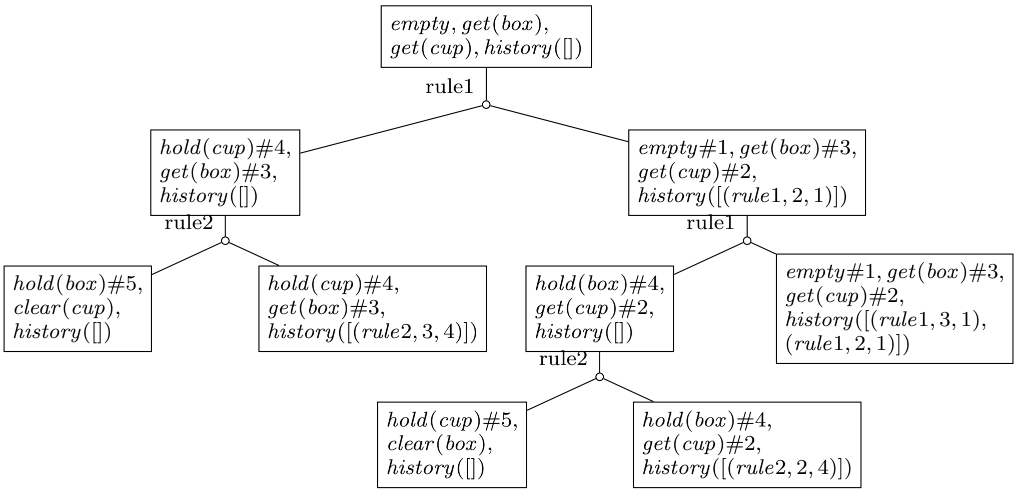
Example 2 (**Blocks World Revisited**)

```
%%%%%%%%%%%%%%%%%%% Extend Constraints %%%%%%%%%%%%%%%%%%%
extend_empty @ empty, id(I) <=> empty_t(I), I1 is I + 1, id(I1).
extend_get @ get(X), id(I) <=> get_t(X,I), I1 is I + 1, id(I1).
extend_hold @ hold(X), id(I) <=> hold_t(X,I), I1 is I + 1, id(I1).
%%%%%%%%%%%%%%%%%%% Transformed Rules %%%%%%%%%%%%%%%%%%%
rule1_t @ get_t(X,ID1), empty_t(ID2), history(L)
```

```
     <=> \+member((rule1,ID1,ID2),L)  | hold(X),history([])
     ; get_t(X,ID1), empty_t(ID2),history([(rule1,ID1,ID2)|L]).

rule2_t @ get_t(X,ID1), hold_t(Y,ID2), history(L)
     <=> \+member((rule2,ID1,ID2),L) | hold(X), clear(Y), history([])
     ; get_t(X,ID1), hold_t(Y,ID2), history([(rule2,ID1,ID2)|L]).
```

Executing the program with the query `‘empty, get(box), get(cup), history([]), id(1)’` produces the tree depicted in Figure 5.



$empty, get(box), get(cup), history([])$
rule1
$hold(cup)\#4, get(box)\#3, history([])$
$empty\#1, get(box)\#3, get(cup)\#2, history([(rule1, 2, 1)])$
rule2
rule1
$hold(box)\#5, clear(cup), history([])$
$hold(cup)\#4, get(box)\#3, history([(rule2, 3, 4)])$
$hold(box)\#4, get(cup)\#2, history([])$
$empty\#1, get(box)\#3, get(cup)\#2, history([(rule1, 3, 1), (rule1, 2, 1)])$
rule2
$hold(cup)\#5, clear(box), history([])$
$hold(box)\#4, get(cup)\#2, history([(rule2, 2, 4)])$

Fig. 5: Derivation tree of Blocks World transformed program. For simplicity a constraint $c(X_1, \ldots, X_n, ID)$ is represented as $c(X_1, \ldots, X_n)\#ID$

As shown in the tree, the derived final states are `‘hold(box),clear(cup)’`,`‘hold(cup), get(box)’`,`‘hold(cup),clear(box)’`, `‘hold(box),get(cup)’`, and `‘empty, get(box), get(cup)’`. These results represent all intermediate and final states from the derivation tree of the Blocks World program. Our proposed transformation derives all states in the derivation tree of any given program because for any CHR state, if a transition is applicable, our program will store the transition information in the transition application history and will make a derivation in which the transition cannot be applied, thus eventually this CHR state will be a final state.

Depending on the application, deriving all intermediate and final states might be unwanted, therefore we present two methods to prune the derived results by the transformed program that correspond to intermediate results in the derivation tree of the original program. The first approach is presented in the previous work presented in (Elsawy et al. 2014). This approach adds pruning rules to the transformed program for every CHR rule in the source program. These rules will be applied only if no other rules are applicable and will prune final states if these states would fire a CHR rule and the transition application history is not checked. A simpler approach is to check the transition application history, if a final state is derived by the transformed program and the transition application history of this state is not empty, then this state corresponds to an intermediate state, because transitions in the transition application history are applicable on this state.

3.3 Evaluation

Initial Goal	*Transformation 1*	*Transformation 2*
empty, get(i1), get(i2)	29	5
empty, get(i1), get(i2), get(i3)	157	16
empty, get(i1), get(i2), get(i3), get(i4)	1169	65
empty, get(i1), get(i2), get(i3), get(i4), get(i5)	11557	326
empty, get(i1), get(i2), get(i3), get(i4), get(i5), get(i6)	141809	1957

Table 1: *Blocks World Comparison (number of results)*

Initial Goal	*Transformation 1*	*Transformation 2*
empty, get(i1), get(i2)	0	0
empty, get(i1), get(i2), get(i3)	0	0
empty, get(i1), get(i2), get(i3), get(i4)	15	0
empty, get(i1), get(i2), get(i3), get(i4), get(i5)	141	0
empty, get(i1), get(i2), get(i3), get(i4), get(i5), get(i6)	1810	47

Table 2: *Blocks World Comparison (runtime in milliseconds)*

In this subsection we evaluate the proposed source-to-source transformation and compare it to the previous source-to-source transformation presented in (Elsawy et al. 2014) in terms of performance. Two metrics are used to show the difference in performance between the two approaches. The first metric is the number of results derived by the transformed programs, while the second metric is the total runtime in milliseconds of the transformed programs. Transformation 1 represents the proposed source-to-source transformation in (Elsawy et al. 2014), while Transformation 2 is the transformation presented in this paper.

Table 1 shows that the proposed source-to-source transformation produces less number of results compared to the source-to-source transformation proposed in (Elsawy et al. 2014). Similarly, Table 2 shows a dramatic decrease in the time taken to fully explore the derivation tree of transformation 1. The used metrics show that the source-to-source transformation proposed in this paper performs better than the source-to-source transformation proposed in (Elsawy et al. 2014).

4 Application

One interesting application for the exhaustive execution of CHR, is that in a sense it brings CHR closer to its declarative form. One can utilize the exhaustive execution to solve complex problems by writing simpler code. For example to find all paths from source to destination in a graph, one could write a CHR program that finds a single path from source to destination. Then the exhaustive program would automatically find all paths, by fully exploring all derivations.

For the graph in Figure 6, there are two paths from node `b` to `f`. With exhaustive

CHR execution, one can write a program that declaratively defines a single path from source to destination, then the exhaustive transformation yields all paths.

A graph can be denoted using `edge/2` constraints representing directed edges from the first argument to the second. Furthermore, nodes containing no outgoing edges are indicated by `final/1` constraints. The simple CHR program that searches for a path from `X` to `Y` through a `search/2` constraint, would produce `path/2` constraints for the traversed path and then a `found/0` constraint is added to indicate the success of the path.

Example 3 (**All Paths**)

```
found    @ search(X,Y), edge(X,Y) <=> path(X,Y), found.
traverse @ search(X,Y), edge(X,Z) <=> path(X,Z), search(Z,Y).
notfound @ search(X,_), final(X) <=> fail.
```

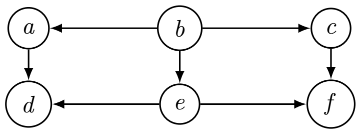


Fig. 6: Graph represented by the constraints '`edge(b,a), edge(b,c), edge(b,e), edge(a,d), edge(e,d), edge(c,f), edge(e,f), final(d), final(f)`'.

Searching for a path from *b* to *f* can be achieved through: '`search(b,f),edge(b,a), edge(b,c),edge(b,e),edge(a,d),edge(e,d),edge(c,f),edge(e,f),final(d),final(f)`'. By executing the program presented above, we will derive the path '`path(b,c), path(c,f),found`' only. Applying the source-to-source transformation presented in this work, and executing the transformed program would yield all paths; i.e. '`path(b,e), path(e,f),found`' and '`path(b,c),path(c,f),found`'. The exhaustive program is obtained using the transformation presented in this work, however the listing of the transformed program is omitted due to space limitations.

5 Conclusion

The paper presented a new approach for the exhaustive execution of CHR programs. The approach involves a source-to-source transformation which expresses any CHR program as one utilizing disjunction, hence forcing an exhaustive explorative execution strategy. The transformation exhaustively traverses the search space, which is created without duplicating derivation paths. This transformation is more efficient than the previous approach of (Elsawy et al. 2014), as was shown in the evaluation.

As future work, we intend to explore different search strategies for the derivation tree which could lead to implementing intelligent search strategies for any CHR program. Due to the disjunctive rule bodies, the transformation can also be easily extended to breadth-first transformation of (De Koninck et al. 2006). Moreover, the transformation can be performed for probabilistic CHR rules. The exhaustive execution will then accumulate probabilities along each traversal path to provide all final states with all their probabilities of occurrence. This can be useful for probabilistic agent-oriented programming.

Appendix - Soundness and Completeness of the Transformation

For simplicity of the proof, we generalize the transformation of any rule *(ruleid* `@` H_k `\` $H_r \Leftrightarrow$ *Guard* `|` *B)* to the following:

$$\begin{array}{c} \mathit{ruleid}\ @\ H_k, H_r, \mathit{trans\_history}(L), \mathit{prop\_history}(P) \Leftrightarrow\ t \notin L, t \notin P, \mathit{Guard}\ \mid \\ B, H_k, \mathit{trans\_history}(\{\}), \mathit{prop\_history}(P \cup \{t\}) \vee \\ H_k, H_r, \mathit{trans\_history}(L \cup \{t\}), \mathit{prop\_history}(P) \end{array}$$

where t is the applied transition.

Lemma 1
Let S be a state derived by the source program P and S' be a CHR state derived by the transformed program P^T such that $S' = S \wedge \mathit{trans\_history}(T) \wedge \mathit{prop\_history}(P)$, let t be some transition applicable on state S', according to the transformation rule defined above, applying t on state S' will derive:

$$S' \mapsto_t (S'_t \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(P \cup \{t\})) \vee (S \wedge \mathit{trans\_history}(T \cup \{t\}) \wedge \mathit{prop\_history}(P)).$$

where $S'_t = S_t$ if transition t is applied on S; $S \mapsto_t S_t$

Definition 1
Let P be a CHR program and P^T be the transformed program of P, let S be a CHR state derived by program P from the initial goal G and S' be a CHR state derived by program P^T from the initial goal $(G \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(\phi))$. Let E_3 be the property defined on S and S', such that $E_3(S, S')$ holds iff $S' = (S \wedge \mathit{trans\_history}(\_) \wedge \mathit{prop\_history}(\_)$

Theorem 1 (*Soundness*)
Given a CHR program P, its corresponding transformed program P^T, and an initial query G. P^T is sound with respect to P if and only if for every CHR state S' derived from the initial goal $(G \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(\phi)) \mapsto^* S'$ by program P^T, there exists a CHR state S derived by program P such that $(G \mapsto^* S)$ and $E_3(S, S')$ holds.

Proof
Base Case. For initial goals G' and G, where $G' = G \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(\phi)$, $E_3(G, G')$ holds.
Hypothesis. Assume that S' and S are two CHR states derived by programs P^T and P respectively and $E_3(S, S')$ holds.
Induction step. We prove that for any transition t applicable on S', such that $S' \mapsto_t S_t^{'}$, then there exists a state S_t derived by P, such that $E_3(S_t, S_t^{'})$ holds.

- From the hypothesis: $S' = (S \wedge \mathit{trans\_history}(\_) \wedge \mathit{prop\_history}(\_))$
- Since every rule in P^T with rule head $H, \mathit{trans\_history}/1, \mathit{prop\_history}/1$ has a corresponding rule in P with rule head H, then if transition t applicable on state S', then t is applicable on state S.

- Applying t on state S, yields $S \mapsto_t S_t$
- Applying t on state S', yields $S' \mapsto_t (S_t \wedge \mathit{trans\_history}(\_) \wedge \mathit{prop\_history}(\_)) \vee (S \wedge \mathit{trans\_history}(\_) \wedge \mathit{prop\_history}(\_))$.
- $E_3(S_t, S_t \wedge \mathit{trans\_history}(\_) \wedge \mathit{prop\_history}(\_))$ holds and $E_3(S, S \wedge \mathit{trans\_history}(\_) \wedge \mathit{prop\_history}(\_))$ holds.

$\square$

Definition 2
Let P be a CHR program and P^T be the transformed program of P, let S be a CHR state derived by program P from the initial goal G and S' be a CHR state derived by program P^T from the initial goal $(G \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(\phi))$. Let E_4 be the property defined on S and S', such that $E_4(S, S')$ holds iff $S' = (S \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'}))$ and $H_S = H_{S'}$ where H_S is the propagation history store of S.

Theorem 2 (*Completeness*)
Given a CHR program P, its corresponding transformed program P^T, and an initial query G. P^T is complete if and only if for every CHR state S derived from the initial goal $G \mapsto^* S$ by program P, there exists a CHR state S' derived by program P^T such that $(G \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(\phi)) \mapsto^* S'$ and $E_4(S, S')$ holds.

Proof
Base case. Initially $G' = (G \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(\phi))$, where G' and G are the initial goals of programs P^T and P respectively. Since $H_G = \phi$, then $E_4(G, G')$ holds.
Hypothesis. Assume that S' and S are two CHR states derived by programs P^T and P respectively and $E_4(S, S')$ holds.
Induction Step. We prove that for any transition t applicable on S, such that $S \mapsto_t S_t$, then there exists a derivation from S' such that $S' \mapsto^* S'_t$ and $E_4(S', S'_t)$.

- From hypothesis: $S' = (S \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'}))$ and $H_S = H_{S'}$
- Since every rule in P with rule head H has a corresponding rule in P^T with rule head $H, \mathit{trans\_history}/1, \mathit{prop\_history}/1$, since the transition history of S' is empty, and $H_S = H_{S'}$, then $t \in \mathit{transitions}(S')$, where $\mathit{transitions}(S')$ is the set that contains all applicable transitions on state S'.
- The transformed program will apply transition t' from $\mathit{transitions}(S')$:
  $S' \mapsto_{t'} (S'_{t'} \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'} \cup \{t'\})) \vee (S \wedge \mathit{trans\_history}(\{t'\}) \wedge \mathit{prop\_history}(H_{S'}))$
  - Case $t = t'$: then $S'_{t'} = S_t$, and since $H_{S_t} = H_{S'} \cup \{t'\}$ then $E_4(S_t, S'_{t'} \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'} \cup \{t'\}))$ holds.
  - Case $t \neq t'$: For this case we will check the second disjunct from applying transition t' on S': $S' \mapsto_{t'} (S \wedge \mathit{trans\_history}(\{t'\}) \wedge \mathit{prop\_history}(H_{S'}))$
    - since $S' = (S \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'}))$, then:
      $(S \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'})) \mapsto_{t'} (S \wedge \mathit{trans\_history}(\{t'\}) \wedge \mathit{prop\_history}(H_{S'})$
    - Thus CHR will apply one or more transitions from $\mathit{transitions}(S')$:
      $(S \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'})) \mapsto^+ (S \wedge \mathit{trans\_history}(T) \wedge \mathit{prop\_history}(H_{S'}))$ where $T \subset \mathit{transitions}(S')$
    - Eventually CHR will apply transition t, since $t \notin T$: $(S \wedge \mathit{trans\_history}(T) \wedge \mathit{prop\_history}(H_{S'})) \mapsto_t S_t \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'} \cup \{t\})$, then $E_4(S_t, S_t \wedge \mathit{trans\_history}(\phi) \wedge \mathit{prop\_history}(H_{S'} \cup \{t\}))$ holds.

$\square$

References

Abdennadher, S. 1997. Operational semantics and confluence of constraint propagation rules. In *Proceedings of the 3rd Intl. Conf. on Principles and Practice of Constraint Programming*. 252–266.

Abdennadher, S. and Schütz, H. 1998. CHR$^{\vee}$: A flexible query language. In *Flexible Query Answering Systems*. Lecture Notes in Computer Science, vol. 1495. Springer-Verlag, 1–14.

De Koninck, L., Schrijvers, T., and Demoen, B. 2006. Search strategies in CHR(Prolog). In *Proceedings of 3rd Workshop on Constraint Handling Rules*. K.U.Leuven, Technical report CW 452, 109–124.

Duck, G. J., Stuckey, P. J., and Sulzmann, M. 2007. Observable confluence for constraint handling rules. In *Logic Programming*. Lecture Notes in Computer Science, vol. 4670. 224–239.

Elsawy, A., Zaki, A., and Abdennadher, S. 2014. Exhaustive execution of CHR through source-to-source transformation. In *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, M. Proietti and H. Seki, Eds. Vol. 8981. Springer, 59–73.

Frühwirth, T. 2009. *Constraint Handling Rules*. Cambridge University Press.

Frühwirth, T., Brisset, P., and Molwitz, J.-R. 1996. Planning cordless business communication systems. *IEEE Expert: Intelligent Systems and Their Applications*, 50–55.

Frühwirth, T. and Raiser, F., Eds. 2011. *Constraint Handling Rules: Compilation, Execution, and Analysis*. Books on Demand.

Koninck, L. D., Schrijvers, T., and Demoen, B. 2008. A flexible search framework for CHR. Vol. 5388. Springer, 16–47.

Lam, E. S. and Sulzmann, M. 2006. Towards Agent Programming in CHR. In *CHR'06: Proceedings of 3rd CHR Workshop*. 17–31.

Martinez, T. 2011. Angelic CHR. In *CHR'11: Proceedings of the 8th Workshop on Constraint Handling Rules*. 19–31.