Towards the Implementation of a Source-to-Source Transformation Tool for CHR Operational Semantics

Ghada Fakhry(✉), Nada Sharaf, and Slim Abdennadher

Computer Science and Engineering Department,
The German University in Cairo, Cairo, Egypt
{ghada.fakhry,nada.hamed,slim.abdennadher}@guc.edu.eg

Abstract. Constraint Handling Rules (CHR) is a high-level committed-choice language based on multi-headed and guarded rules. Over the past decades, several extensions to CHR and variants of operational semantics were introduced. In this paper, we present a generic approach to simulate the execution of a set of different CHR operational semantics. The proposed approach uses source-to-source transformation to convert programs written under different CHR operational semantics into equivalent programs in the CHR refined operational semantics without the need to change the compiler or the runtime system.

Keywords: Source-to-Source Transformation · Constraint Handling Rules · Operational Semantics

1 Introduction

Constraint Handling Rules (CHR) [4] is a high level language that was introduced for writing constraint solvers. CHR is a committed choice language based on multi-headed and guarded rules. With CHR, users can have their own defined constraints. CHR transforms constraints into simpler ones until they are solved. Over the past decade, CHR has matured into a general purpose language. In addition, the number of CHR extensions and variants has increased [10]. These extensions have operational semantics different than the refined operational semantics (w_r) of CHR [3] regarding some properties like execution control, expressivity and declarativity.

Such extensions tackle some weaknesses and limitations of CHR and offer interesting properties to its users [10]. However, users cannot use these extensions directly through SWI-Prolog [13] since their operational semantics is different than the refined operational semantics supported by SWI-Prolog (w_r). Some extensions, nevertheless, provide transformation schemes to the refined operational semantics. However, such schemes usually require accessing the compiler and using additional low level tools [6,8,11,12] .

G. Gupta and R. Peña (Eds.): LOPSTR 2013, LNCS 8901, pp. 145–163, 2014.
DOI: 10.1007/978-3-319-14125-1_9

The work presented here extends the tool presented in [2] which enhanced CHR with visualization features through a source-to-source transformation approach without changing the compiler or the runtime system.

The aim of this work, on the other hand, is to introduce an approach that is able to automatically simulate the execution of a set of CHR operational semantics. Such operational semantics could have different execution models than the refined operational semantics. The proposed approach uses source-to-source transformation to convert CHR programs written under different operational semantics to equivalent programs that could be used with the refined operational semantics (w_r). This process does not require any changes to the compiler or the runtime system. The paper presents the general scheme that could be used with different operational semantics. In addition, the scheme is applied on a set of the existing CHR operational semantics. Although previous approaches provided transformation techniques, the focus here is on achieving a general approach that is usable without having to change any details regarding the runtime system. The presented work thus does not aim to provide a more efficient alternative but rather a more general one.

The paper is organized as follows. Section 2 briefly discusses the syntax and semantics of Constraint Handling Rules. Section 3 introduces the general transformation approach and the structure of the transformed file. In addition, it introduces the implementation of an explicit propagation history. Section 4 shows how the transformation approach is applied to implement a set of different CHR operational semantics. Section 5 provides an sketch proof for the equivalence between source programs and the transformed programs. Finally, we conclude with a summary and a discussion of future work in Section 6.

2 Constraint Handling Rules

This section introduces the syntax of CHR. In addition, we informally explain the abstract semantics and the refined operational semantics.

2.1 Syntax

CHR programs consist of a set of guarded rules that are applied until a fixed point is reached. In CHR two types of constraints are available. The first type is the built-in constraints provided through the host language. The second type of constraints is the CHR or user-defined constraints that are defined through the rules of a CHR program [5]. A CHR program consists of a set of a so-called "simpagation" rules in the following format:

$$H^k \ \setminus \ H^r \ \Leftrightarrow \ G \ \mid \ B.$$

The head of the CHR rule, which comes before the ($\Leftrightarrow$), consists of a conjunction of CHR constraints only. The elements of H^k are the constraints that are kept after the rule is executed. On the other hand, the constraints in H^r are removed after executing the rule. G is the optional guard that consists of built-in constraints. The body (B) could contain both CHR and built-in constraints. Two other types of rules exist. They are considered as special cases of simpagation rules. The first one occurs whenever H^k is empty. Such rules are called "simplification" rules. In this case, the head of the rule consists only of CHR constraints that should be removed on executing the rule. Such rules replace constraints by simpler ones. A simplification rule thus has the following format:

$$H^r \;\Leftrightarrow\; G \;\mid\; B.$$

The second rule type is propagation rules. In a propagation rule, H^r is empty. Consequently, all head-constraints are kept after the rule is executed adding the constraints in the body to the constraint store. This may cause further simplification afterwards. Propagation rules have the following format:

$$H^k \;\Rightarrow\; G \;\mid\; B.$$

2.2 Operational Semantics

The operational semantics of CHR programs is defined by a state transition system. The complete definition of state transitions of the abstract semantics and the refined operational semantics are introduced in [5] and [3] respectively. The execution of a CHR program starts from an initial state. Rules are applied until a final state is reached. A final state is a state where no more rules are applicable.

A rule is applied if the constraints in store matches the head-constraints of the rule, and the guard of the rule succeeds. The current implementation of CHR in SWI-Prolog does not allow binding variables in the guards of rules by default [13]. According to the type of the rule, the constraints that matched the head-constraints of the rule are either removed or kept after the rule application.

Consider the following example that computes the minimum of a multiset of numbers. The numbers are represented by the CHR constraint `min/1` whose argument is the value of the number.

```
find_min @ min(N) \ min(M) <=> N=<M | true.
```

The initial query for the program is a multiset of constraints representing the numbers whose minimum is to be computed. Each time the simpagation rule (`find_min`) is applied, two numbers are compared and the larger one is removed. This rule is applied exhaustively until the constraint store contains one `min` constraint representing the minimum number.

The abstract CHR operational semantics does not specify the order in which the constraints of the initial goal are processed, or the order of the application of the rules. For the initial goal `min(1)`, `min(3)`, `min(0)`, `min(2)`, the result is always `min(0)`. However, many execution paths could be taken. The computation can start with applying the rule on the constraints `min(1)` and `min(3)`, or the constraints `min(1)` and `min(0)`, or any other combination of two constraints matching the head of the rule. In addition, in case the goal constraints match the heads of more than one rule, then any of them could be applied. In other words, there is no restriction on the order of application of rules.

The refined operational semantics fixes, in part, the non-determinism of the abstract semantics. It fixes the order of processing the goal constraints and the order in which the rules are applied. In the refined operational semantics, goal constraints are processed from left to right, and rules are tried in the textual order of the program. Accordingly, for the same goal in the above example, the simpagation rule will be applied first with `min(1)` and `min(3)` constraints then it will continue with the remaining constraints.

In both semantics, there is a restriction for the application of propagation rules. A propagation rule is allowed to be applied only one time with the same combination of constraints that matched the head-constraints of the rule.

3 The Transformation Approach

This section introduces a new source-to-source transformation approach that is able to transform CHR programs written under different operational semantics into CHR programs that are equivalent when executed under the refined operational semantics.

Building on the representation used in [2], the rules of the source program are transformed into a so-called “relational normal form” introduced in [7]. This normal form uses special CHR constraints that represent the components of a rule. For example, the rule `find_min` in Section 2.2 is represented in the relational normal form as follows:

```
head(find_min,‘min(N)’,keep),
head(find_min,‘min(M)’,remove),
guard(find_min, ‘N<=M’),
body(find_min,‘true’)
```

The CHR solver is first parsed. The parser extracts the information of the program and represents it in the normal form. The transformer is a CHR solver that runs on the relation normal formal of the source program and writes the new rules into the transformed program file.

The idea of the presented transformation approach relies on the execution model of the operational semantics. The work in this paper involves transforming different operational semantics that have different state transitions for rule choice and rule application. In addition, inverse execution of the rules of a program [14] is also considered. The transformation allows for the simulation of different rule-choice and rule-application state transitions of different operational semantics. This is done by separating rule matching and rule application into two steps. The basic idea is to delay the application of the body of the rule. With this approach, rather than having to apply the first matched rule, the new program is able to choose from the set of all the applicable rules. The adopted candidate set resolution approach is similar to the conflict resolution mode introduced in [5].

However, the presented work provides an automated transformation methodology that is able to combine the different properties of the execution models of the operational semantics. As a result, the execution of different operational semantics is possible. Two rules are generated for every rule in the source program. The first is a propagation rule that replaces the body by a new CHR constraint representing the rule name and the constraints that matched the rule head. The second rule applies the (possibly modified) body of the rule. The choice of the rule to be applied depends on the candidate set resolution strategy. In the current implementation, the transformer provides different resolution strategies simulating different rule-choice state transitions. The execution model of the operational semantics was represented through a set of properties. Such properties encode the execution direction, the candidate set resolution strategy and whether multiple-rules matching is allowed. Such property-set is then used to construct the transformed program.

Through specifying the properties of the execution model, the proposed approach is able to transform different operational semantics. The proposed transformation allows forward or inverse rule application as execution strategies. It also offers a set of candidate set resolution strategies. In addition, at each computation step, a choice of single or multiple rules matching is possible.

Section 3.1 explains the transformation of the rules according to the properties of the execution model. Section 3.2 introduces the idea of implementing an explicit propagation history in the transformed program.

3.1 The Transformed Program Structure

Figure 1 shows the steps of constructing the transformed program. The choice between the different construction paths depends on properties of the execution of the semantics. The construction of the transformed file is done in four steps as explained below.

1. The first step adds for every CHR constraint `c(X)` in the source program, a simplification rule (`extend`) in the transformed program P^T in the form:

   ```
   extend @ c(X) <=> c(X,_).
   ```

   This way, when executing the transformed program, an extended CHR constraint `c(X,V)` is created for each CHR constraint `c(X)` similar to the approach used in [8]. `V` is a fresh Prolog variable used as an explicit identifier for a constraint and is also used in the implementation of the propagation history explained in Section 3.2. In addition, all the constraints in the heads of the rules of the transformed file are extended with an additional argument. This argument represents the unique identifier of the constraint.
2. The second step adds, for each rule in the source program, a propagation rule that differs according to execution strategy of the operational semantics. One of two possible rules is added. The rule `Fmatch` is added in the case of forward execution. On the other hand, the rule `Imatch` is added whenever inverse execution is needed.

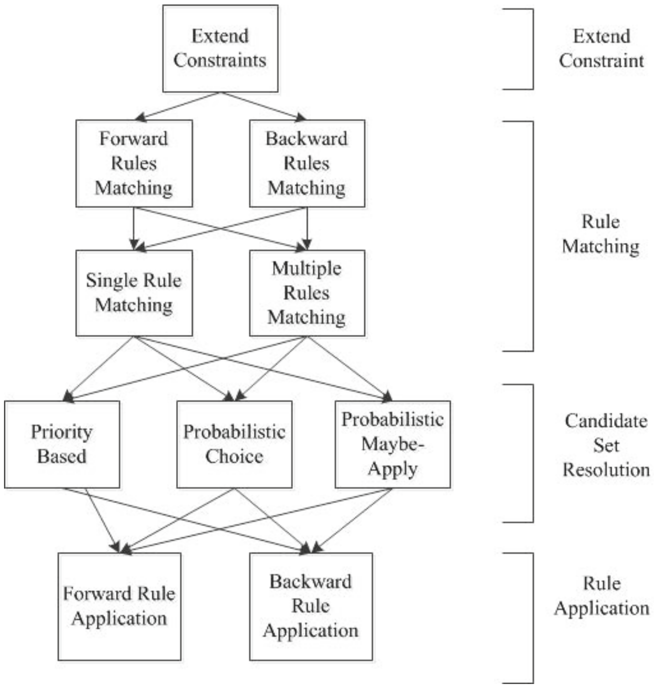

Fig. 1. Construction Steps of The Transformed File

Although both rules are propagation rules, they have different constituents. A forward match (`Fmatch`) rule has the same head-constraints and guard as the original rule of the source program rule. However, the head of an inverse match (`Imatch`) rule contains the kept head-constraints (if any) in addition to the CHR constraints of the body of the original rule. If the body of the source program rule contains built-in constraints then they are added to the guard of this new rule. The body of `Fmatch` and `Imatch` is a new CHR constraint `cand/3`. The arguments of this new constraint are the rule name, the list of identifiers of the head-constraints od the rule in addition to a number that could represent some specific property of the rule. For example, in the case of CHR with user-defined rule priorities, this number represents the priority of each rule. Since inverse execution of CHR rules is a one-to-many relationship, a `true` disjunct is added to allow backtracking for more than one result.
Thus, for every CHR rule in the source program P:

$$\texttt{ri} \ @\ H^k \backslash H^r \ \Leftrightarrow\ G \mid B.$$

we will have one of the following rules in the transformed program P^T:

$$\texttt{Fmatch-ri} \ @ \ H^r, H^k \ \Rightarrow \ G \mid cand(ri, Ids, p).$$
$$\texttt{Imatch-ri} \ @ \ H^k, B \ \Rightarrow \ G \mid cand(ri, Ids, p) \ ; \ true.$$

Whenever a rule is applicable, then a new constraint `cand/3` is created for the applicable rule. This new constraint is added to the constraint store. The new `cand/3` constraint means that the applicable rule could be fired with this specific combination of constraints. If the operational semantics allows multiple-rules matching at each computational step, an additional constraint `id/1` is added to the head-constraints of the matching rules (`Fmatch` or `Imatch`). Changing the value of the argument of `id/1` allows the propagation rules to be matched with the same instances of constraints for multiple times. This behaviour is needed to reach the correct output. An example of this is the case of CHR with user-defined rule priorities, where the highest priority rule, among the applicable rules, is fired at each computational step. The rules that were not fired at one step due to the existence of higher priority rule(s), should be given a second chance of application. However, the match (`Fmatch` or `Imatch`) rules are propagation rules that are fired for a specific combination of constraints once. Thus changing any argument of the constraints of the heads of such rules allow them to be fired again (i.e. giving the rest of the rules another chance). Since the original constraints cannot be manually modified, using the auxiliary constraint `id/1` with an argument that changes with the rule application solved this problem.

In addition, a propagation rule (`trigger`) is added at the end of the matching rules. This new rule adds the CHR constraint `start/0` to trigger the candidate set resolution step. The constraint `trigger/0` is added to the end of the constraints in the original query to ensure that it is only activated at the end.

```
trigger @ trigger, id(Ni) ==>  start.
```

On executing the transformed program, the result of this step is a set of all the applicable rules. Each of the candidate rules is represented by the constraint `cand/3`. In the refined operational semantics, head-constraints are searched from left to right. However, for simpagation rules, the head-constraints to be removed are tried before the constraints to be kept [5]. Thus to preserve the same order, removed head-constraints are added before the kept head-constraints in the `Fmatch` transformed rules. `p` is a property specified by the operational semantics. In the current implementation, the property is concatenated to the rule name instead of using the directive `pragma` argument to give hints to the compiler.

3. In the third transformation step, rules are added to perform candidate set resolution. Only one of the candidate applicable rules is chosen to be applied according to the resolution strategy. In the current implementation of the transformer, this step is customized according to the respective CHR operational semantics. The current implementation allows probabilistic and priority based resolution strategies. More details about candidate set resolution is given in Section 4 with examples. The result of this step is the rule chosen to

be applied represented by a new CHR constraint `fire/2`. The first argument of `fire/2` is the rule name and the second argument is a list of identifiers of the head-constraints of the rule. The list of constraint identifiers is added to ensure that the rule will only be applied with this specific combination of constraints.

4. The last part in the transformed file is responsible of the actual rule application. A new rule is added for each rule in the original solver. The new rule is chosen according to the execution strategy. Thus either a forward application (`Fapply`) or an inverse application (`Iapply`) rule is added to the transformed program. In `Fapply` rules, a new constraint `fire/2` is added to the head-constraints to be removed of the source program rule (if any). This way ensures that only this rule will be applied with the specific combination of constraints in the list. The body of the rule remains unchanged. In `Iapply` rules, the `fire/2` constraint and the body of the original rule are added as head-constraints to be removed. The body of the `Iapply` rule contains the head-constraints that should be removed.
   For every CHR rule in the source program P:

$$\texttt{ri}\ @H^k \backslash H^r \Leftrightarrow G \mid B.$$

   we will have one of the following two rules in the transformed program P^T:

$$\begin{array}{l} \texttt{Fapply-ri}\ @\ H^k\backslash\ \texttt{fire(ri,Ids)}\ , H^r \Leftrightarrow B. \\ \texttt{Iapply-ri}\ @\ H^k\backslash\ \texttt{fire(ri,Ids)}\ , B \Leftrightarrow H^r. \end{array}$$

The rules are written such that the execution of one rule in the source program is done in 4 steps in the transformed program The execution of the transformed program starts by extending the original query constraints. The extended constraints (constraints with the additional identifier argument) then try to match the propagation rules in the second part of the transformed file. Among the set of the applicable rules represented by `cand/3` constraints, one rule is chosen according to the candidate set resolution strategy. Finally, the chosen rule is applied. Execution then proceeds by extending any constraints added after applying the chosen rule. The new constraints then try to match the propagation rules. Candidate set resolution is applied afterwards on the new candidate-set and so on until reaching a fixed point where no more rules are applicable. The new file depends on the textual order of the rules since it runs using the refined operational semantics implemented in SWI Prolog.

3.2 Propagation History Implementation

In some of the cases, the set of matched rules that were not applied at one computation step are given a second chance in the next computation step. This is the case when the operational semantics allows multiple-rules matching at each computation step and one rule application. In order to allow for multiple-rules matching, the constraint `id/1` is added to the head-constraints of the transformed rules.

However, this approach raised a problem with propagation rules. If a propagation rule was chosen to be fired then in the next computation step the same propagation rule will also be applicable, because the constraints that matched the head were not removed. This causes a problem of trivial non-termination in the transformed program.

This problem would be solved if every propagation rule is fired only once for each specific combination of constraint identifiers. This was done by implementing an explicit propagation history. In the proposed approach, a new constraint `history/1` is added to the original query. The argument of `history/1` is a list that contains a set of tuples `(r,I)`. Initially, the list is empty. The first argument (`r`) is the propagation rule name while the second argument (`I`) is an ordered list of the identifiers of constraints that matched the head-constraints of rule `r`. The size of the propagation history depends on the propagation rules in the program. At any computation step the length of the list of `history/1` corresponds to the number of fired propagation rules.

In the transformed program, propagation rules are modified to be applied only if the tuple containing the rule name and the list of identifiers of constraints does not exist in the propagation history, since each constraint in the executed program has a unique identifier. For example, the following rule in the original program P:

```
rule1 @ a(X) ==> b(X).
```

will have the corresponding propagation rule transformed program P^T:

```
match-rule1 @ a(X,Id1), history(L) ==> \+member((rule1,[Id1]),L)
                                        | cand(rule1,[Id1],1).
```

Moreover, in this example, the property specified to the operational semantics is the rule order. Thus for the rule `rule1`, this property is set to `1` since the program contains only one rule.

In addition, if a propagation rule is chosen to be applied, a new tuple with the name of the rule and a list with the identifiers of the matched constraints is added to the propagation history. Thus, the propagation rules are modified in the rule application part to update the propagation history. The rule in the previous example generates the following simpagation rule in the transformed program P^T:

```
apply-rule1 @ a(X,Id1) \ fire(rule1,[Id1]), history(L)
                        <=> b(X),history([(rule1,[Id1])|L]).
```

4 Source-to-Source Transformation for Different CHR Operational Semantics

This section shows, through examples, how the presented transformation approach is applied to a set of different CHR operational semantics. Transformation for Probabilistic Constraint Handling Rules is explained in Section 4.1, CHR with user defined-rule priorities is introduced in Section 4.2. Section 4.3 introduces transformation for CHRiSM. Finally, transformation for inverse CHR is introduced in Section 4.4.

4.1 Transformation for Probabilistic Constraint Handling Rules

Probabilistic Constraint Handling Rules (PCHR) [6] is an extension of CHR that allows for a probabilistic rule choice among the applicable rules. The choice of the rule is performed randomly by taking into account the relative probability associated with each rule. PCHR modifies the CHR abstract semantics (w_t) in the "Apply" transition by specifying the probability of the choices of the rules. This results in an explicit control of the chance that certain rules are applied according to their probabilities. The "Apply" transition of w_t chooses a rule from the program for execution. Constraints matching the head of the rule should exist in the store. In addition, the guard should be satisfied. PCHR rules are the same as CHR rules but with the addition of a number representing the relative probability of each rule.

PCHR is implemented using the proposed approach with forward execution strategy, multiple-rules matching. Candidate set resolution is done through a random choice after normalizing the probabilities of the probabilistic rules. The following example shows a PCHR program [6] that generates a n bit(s) random number. The number is represented as binary list of n bit(s). The list is generated bit by bit recursively and randomly. As long as N is greater than zero, the next bit will be either 0 or 1 by applying either the second or the third rules with equal probability; otherwise the non-probabilistic rule `r1` will be applied and the recursion ends. The program is:

```
r1 @ rand(N,L) <=> N =:= 0 | L = [].
r2_50 @ rand(N,L) <=> N>0 | L=[0|L1], N1 is N-1, rand(N1,L1).
r3_50 @ rand(N,L) <=> N>0 | L=[1|L1], N1 is N-1, rand(N1,L1).
```

The transformation will result in the following program:

```
extend @ rand(V2,V1) <=> rand(V2,V1,_).

r1 @ rand(N,L,Id0) <=> N =:= 0 | L = [].
match-r2_50 @ rand(N,L,Id0),id(Ni)==>N>0 | cand(r2_50,[Id0],50).
match-r3_50 @ rand(N,L,Id0),id(Ni)==>N>0 | cand(r3_50,[Id0],50).
trigger @ trigger, id(Ni) ==>  start.

start     @ cand(R,IDs,N),start <=> random(0,100,Random),
                                    cand(R,IDs,0,N,N,Random).
normalize @ cand(R,IDs,N,M,UB,Random),cand(R1,IDs2,N1)
                                <=> M2 is M+N1,UB2 is UB+N1,
                                    cand(R1,IDs2,M,M2,UB2,Random),
                                    cand(R,IDs,N,M,UB2,Random).
drop   @ id(Ni)\ cand(R,IDs,M,M1,100,Random) <=> Random<M  | true.
drop   @ id(Ni)\ cand(R,IDs,M,M1,100,Random) <=> Random>=M1| true.
choose @ cand(R,IDs,M,M1,100,Random), id(Ni)
                              <=> M=<Random,Random<M1
                                 |fire(R,IDs),Ni2 is Ni+1,id(Ni2).
apply-r2_50 @ fire(r2_50,[Id0]),rand(N,L,Id0)
                              <=> N>0 | L=[0|L1] , rand(N-1,L1).
apply-r3_50 @ fire(r3_50,[Id0]),rand(N,L,Id0)
                              <=> N>0 | L=[1|L1] , rand(N-1,L1).
```

The rule `start` triggers the probability normalization by replacing the first candidate rule `cand/3` by `cand/6`. The additional arguments are a list of the constraint identifiers that were matched in the head of the rule, the lower bound and the upper bound of the rule probability interval. In addition, the last argument is a random number in the interval from 0 to the sum of all rule probabilities calculated by the built-in Prolog predicate `random/3`. The rule `normalize` keeps on replacing the rest of the candidate rules represented through `cand/3` constraints by the extended constraint `cand/6`, each with the lower and upper bound interval of the corresponding probability of the rule. The arguments of `fire/2` are the rule name and the list of constraints identifiers that were matched in the head. Otherwise, the `cand/6` constraint is replaced by true by the rules `drop` which means that this rule will not be applied.

4.2 Transformation for Constraint Handling Rules with User-Defined Rule Priorities

CHR^{r^p} extends CHR with user-defined rule priorities [8]. Rule priorities improve the expressivity of CHR as they allow for a different choice for rule application depending on the respective rule priority, resulting in a more flexible execution control. The operational semantics w_p for CHR^{r^p} only adds restrictions to the applicability of the "Apply" transition of the abstract CHR semantics w_t. The rest of state transitions are equivalent in both semantics. For CHR^{r^p} programs in which all rule priorities are equal, every execution strategy under w_t is consistent with w_p. Thus, such programs can be executed using the refined operational semantics as implemented by the current CHR implementations.

In [8], a source-to-source transformation approach that uses some of the compiler directives is presented. In [8], constraints are not activated when introduced to the store by the default transitions of the refined operational semantics, which are the "Activate" and "Reactivate" transitions [3]. Instead, they remain passive using the compiler directive `passive/1` and are scheduled for activation with the corresponding rule priority. After trying all the possible matching rules with the constraints in the query, the highest priority scheduled constraint is activated.

CHR^{r^p} is implemented using the proposed approach with forward execution, multiple-rules matching and a rule priority candidate set resolution strategy. The following example [8] shows the difference in the execution of the refined operational semantics and CHR^{r^p}. For the same initial query `a`, the refined operational semantics will apply the rules in the following order: 1,2,4,3. While in CHR^{r^p}, rule 3 has higher priority than rule 4. Therefore, the rules will be applied in the following order: 1,2,3. Rule 4 will not be applied anymore because constraint a is removed by rule 3. The same example is used to illustrate the proposed transformation approach.

```
r1_1 @ a ==> print('rule 1 \n'),b .
r2_2 @ a , b ==> print('rule 2 \n').
r3_3 @ a <=> print('rule 3 \n').
r4_4 @ a , b ==> print('rule 4 \n').
```

The transformation will result in the following program:

```
extend @ a <=> a(_).
extend @ b <=> b(_).

match-r1_1 @ a(Id0),id(Ni),history(L)
            ==> \+ member((r1_1,[Id0]),L)|cand(r1_1,[Id0],1).
match-r2_2 @ a(Id0),b(Id1),id(Ni),history(L)
            ==> \+ member((r2_2,[Id0,Id1]),L)|cand(r2_2,[Id0,Id1],2).
match-r3_3 @ a(Id0),id(Ni) ==> cand(r3_3,[Id0],3).
match-r4_4 @ a(Id0),b(Id1),id(Ni),history(L)
            ==> \+ member((r4_4,[Id0,Id1]),L)|cand(r4_4,[Id0,Id1],4).
trigger @ trigger,id(Ni) ==>  start.

start   @ start  <=> candList([]).
collect @ candList(L),cand(R,IDs,N) <=> candList([(N,R,IDs)|L]).
choose  @ candList(L),id(Ni) <=> sort(L,[(P,H,IDs)|T]),fire(H,IDs),
                                 N2 is Ni+1,id(N2).

apply-r1_1 @ a(Id0)\ fire(r1_1,[Id0]),history(L)
             <=> print('rule 1'), b, history([(r1_1,[Id0])|L]).
apply-r2_2 @ a(Id0),b(Id1)\ fire(r2_2,[Id0,Id1]),history(L)
             <=> print('rule 2'),history([(r2_2,[Id0,Id1])|L]).
apply-r3_3 @  fire(r3_3,[Id0]),a(Id0) <=> print('rule 3').
apply-r4_4 @ a(Id0),b(Id1) \ fire(r4_4,[Id0,Id1]), history(L)
             <=> print('rule 4'),history([(r4_4,[Id0,Id1])|L]).
```

In the transformed program, the rule with the highest priority among the set of applicable rules is chosen to be applied. The three rules start, collect, and choose are added to the transformed program to perform the candidate set resolution according to the priorities of the rules. The rule start initalizes an empty priority list such that the applicable rules represented by cand/3 are added to this list by the collect rule. After all cand/3 constraints are added to the list, the rule choose sorts the list and the rule with the highest priority is chosen for application. The chosen rule is represented by a new CHR constraint fire/2, whose first argument is the rule name and the second argument is a list of identifiers of the head-constraints of the rule. The list of constraint identifiers is added to ensure that the rule will only be applied with the specific combination of constraints. In addition, the argument of constraint is incremented to allow match rules to be tried again.

4.3 Transformation for CHRiSM

CHRiSM is a probabilistic extension of CHR that is based on CHR and PRISM [9]. The main difference between the semantics of CHRiSM and PCHR is that the rule probabilities have a localized meaning. The probability of a rule application does not depend on the other applicable rules. CHRiSM semantics adds two features to CHR [9]. First, a defined probability for the entire rule application given by the "Maybe-Apply" state transition. In the "Maybe-Apply" transition, according to the probability of the rule, it is either applied or not. However, if the rule is not applied, the propagation history is updated to prevent further rule application with the same combination of constraints that matched the rule head. The second feature is the ability to define a probability for each disjunct in the rule body in $\mathsf{CHR}^{\vee}$ [1] given by the "Probabilistic Choice" transition. In "Probabilistic Choice", one disjunct is chosen probabilistically according to its probability relative to the other disjuncts.

In this paper, the transformation for CHRiSM implements programs with a user-defined rule probability for the entire rule application, corresponding to the "Maybe Apply" transition only. In CHRiSM operational semantics, a rule with a probability p means that whenever the rule is applicable, it will only be applied with a probability p. If the rule probability is not defined, it is set to uniform distribution 0.5. CHRiSM is implemented using the proposed approach with forward execution, single-rule matching and a probabilistic rule application choice.

The following example illustrates the transformation of CHRiSM to the refined operational semantics (ω_r). Starting with initial query `a`, it is probable that the first rule is applied. If the first rule is applied, then the constraint `b` will be added to the constraints store. Consequently, there is a chance to apply the second rule with probability 0.5, removing constraint `a` and adding `c` to the constraint store. The program is:

```
r1_50 @ a ==> b .
r2_50 @ b \ a <=> c .
```

The transformation will result in the following program:

```
extend @ a <=> a(_).
extend @ b <=> b(_).
extend @ c <=> c(_).
match-r1_50 @ a(Id0)==>cand(r1_50,[Id0],50).
match-r2_50 @ a(Id0),b(Id1)==>cand(r2_50,[Id0,Id1],50).

start @ cand(R,IDs,N) <=> random(0,100,Random),cand(R,IDs,N,Random).
choose-apply  @ cand(R,IDs,M,Random) <=> Random=<M | fire(R,IDs,1).
choose-ignore @ cand(R,IDs,M,Random) <=> M<Random  | fire(R,IDs,2).

apply-r1_50 @ a(Id0)\ fire(r1_50,[Id0],1) <=>b.
apply-r2_50 @ b(Id1)\ fire(r2_50,[Id0,Id1],1),a(Id0) <=> c.
```

```
ignore-r2_50 @ a(Id0),b(Id1)\ fire(r2_50,[Id0,Id1],2)<=> true.
ignore-r1_50 @ a(Id0)\ fire(r1_50,[Id0],2)<=> true.
```

In CHRiSM, each rule is given one chance for application with every combination of constraints. In order to achieve that, the constraint `id(Ni)` is not added to the head-constraints in the `match` rules similar to PCHR and CHR$^{r^p}$. In addition, since the choice is whether to apply the rule or not, there is only one candidate rule at each computation step, therefore no need to add the rule `trigger`.

Whenever a rule is applicable, the `cand/3` constraint of the applicable rule will fire the rule `start` in the candidate set resolution rules. Similar to "Maybe Apply" transition in the CHRiSM operational semantics [9], the body of the rule gets applied with a probability P. The rule `start` generates a random number between 0 and 1 and replaces `cand/3` with `cand/4`. The additional argument is the randomly generated number. The rules `choose-apply` and `choose-ignore` determine whether the rule will be applied according to the randomly generated number. In both cases, the `cand/4` constraint is replaced by `fire/3` constraint. If the randomly generated number is less than the rule probability, the third argument in `fire/3` is set to 1, otherwise it is set to 2.

For simplification and simpagation rules, if the rule is chosen not to be applied then the removed head-constraints should not be removed from the store. In order to keep the same instances of removed head-constraints in store when the probabilistic rule is not applied, each rule in the source program will have an additional `ignore` rule in the transformed file. The rule `ignore` is a simpagation rule where the head-constraints are added as kept head-constraints. Only the `fire/3` constraint with the last argument set to 2 is to be removed. In addition, the body of the rule is replaced by true.

4.4 Transformation for Inverse Constraint Handling Rules

The execution of traditional CHR starts from the initial state and applies program rules until reaching a fixed point or a final state where no more rules are applicable. Inverse execution of CHR rules starts from a state and applies the inverse of program rules in order to reach the initial state. The "Apply" transition of the inverse CHR is the same as "Apply" transition of the abstract semantics of CHR but with exchanging the left and right hand side states of the transition [14]. Inverse CHR is implemented using the proposed transformation approach with inverse execution of rules, multiple-rules matching and rule priority candidate set resolution, where the rule priority is the textual rule order. Thus, the first rule in the program has the highest priority. However, different resolution strategies could be used. The following example [14] is an exchange source for elements in a list. Elements are represented by constraint `a/2`, the first argument is the index of the element in the list and the second argument is the value of the element.

```
eSort @ a(I,V),a(J,W) <=> I>J , V<W | a(I,W),a(J,V).
```

The transformation will result in the following program:

```
extend @ a(V2,V1) <=> a(V2,V1,_).
Imatch-eSort @ a(I,W,Id0),a(J,V,Id1),id(Ni)
                         ==> I>J,V<W | cand(eSort,[Id0,Id1],1) ; true.
trigger @ trigger, id(Ni) ==>  start.
start   @ start  <=> candList([]).
collect @ candList(L),cand(R,IDs,N) <=> candList([(N,R,IDs)|L]).
choose  @ candList(L),id(Ni) <=> sort(L,[(P,H,IDs)|T]),fire(H,IDs),
                                 N2 is Ni+1,id(N2) ; true.
Iapply-eSort @ fire(eSort,[Id0,Id1]),a(I,W,Id0),a(J,V,Id1)
                        <=> I>J,V<W | a(I,V) , a(J,W).
```

The current implementation of the transformer does not distinguish between user-defined and built-in constraints in reverse execution of programs. Accordingly, the transformation is limited to programs with rules whose body contain user-defined constraints only.

5 Equivalence Proof

In this section, we will show how the newly transformed file program is able to capture the needed operational semantics. In other words, we will introduce how the execution of the rules in the new solver is equivalent to the corresponding semantics. For proof of concept, we will show the equivalence of the execution of the transformed program under w_t[5] with the execution of the original program under w_p for CHR^{rp} [8].

A state in w_t[5] is a tuple in the form $\langle G, S, B, T\rangle_n$. The components of the state are defined as follows: The goal G is a multiset of all unprocessed constraints. The CHR store S is a set of numbered user-defined constraints that can be matched with rules in a given program. B is the built-in constraints store. It is the conjunction of built-in constraints that have been added to the built in constraint store. The propagation history T is a set of tuples (r, I), where r is the rule name and I is a list of identifiers of constraints that were matched in the rule head. Finally, n is a counter representing the next free integer for constraint identifying. For the sake of brevity, only G and S are shown in the proof [5].

w_r[3] provides a deterministic execution strategy for any goal. Therefore, for any program P and an initial state S, every derivation for P, $S \overset{w_r}{\underset{\mathrm{P}}{\longmapsto}}{}^* \mathrm{S}'$ corresponds to a derivation $S \overset{w_t}{\underset{\mathrm{P}}{\longmapsto}}{}^* \mathrm{S}'$. The provided sketch proof is based on the mapping between w_t and w_r.

Theorem 1. *Given a CHR program P and its corresponding transformed program $T(P)$ and two states $S_1 = \langle G, \phi\rangle$ and $S_2 = \langle G \bigcup Aux, \phi\rangle$ where G contains the initial goal constraints and Aux is a set of auxiliary constraints. Then the following holds:*

If $S_1 \overset{w_p}{\underset{P}{\longmapsto}}{}^* {S_1}'$ and $S_2 \overset{w_t}{\underset{T(P)}{\longmapsto}}{}^* {S_1}' \cup Aux'$ then $T(P)$ is equivalent to P.

Proof. (Sketch)

Table 1 shows the computational steps of executing one rule of the program P under w_p starting with S_1. On the other hand, tables 2 to 5 show the computational steps of executing the transformed program $T(P)$ under w_t starting with S_2.

Table 1. Computation steps under w_p

	$\langle G_1, \phi \rangle$	G contains initial query constraints
$\xmapsto[\text{introduce}]{}^*$	$\langle \phi, S \rangle$	G The store S contains all the activated goal constraints and the goal G is empty
$\xmapsto[\text{apply}]{}^*$	$\langle G, S \rangle$	G contains the added constraints after the rule application (if any)

Table 2. Computation of step 1 (Constraints Extending)

	$\langle G \bigcup Aux, \phi \rangle$	Aux contains `trigger` and `id(1)` constraints, G contains initial query constraints
$\xmapsto[\text{introduce}]{}^*$	$\langle \phi, S \rangle$	The store S contains all the activated goal constraints and the auxiliary constraints
$\xmapsto[\text{apply extend}]{}^*$	$\langle G', S \rangle$	G' contains extended constraints after applying the `extend` rules on the initial query constraints
$\xmapsto[\text{introduce}]{}^*$	$\langle \phi, S \rangle$	The store S contains all the activated extended constraints in addition to the auxiliary constraints

Table 3. Computation of step 2 (Rule Matching)

$\xmapsto[\text{apply match}]{}^*$	$\langle G'', S \rangle$	G'' contains `cand/3` constraints after applying the `match` rules
$\xmapsto[\text{introduce}]{}^*$	$\langle \phi, S \rangle$	S contains all the activated `cand/3` constraints
$\xmapsto[\text{apply trigger}]{}$	$\langle \{start\}, S \rangle$	The goal contains only one constraint(`start`) since it is the only applicable rule
$\xmapsto[\text{introduce}]{}$	$\langle \phi, S \rangle$	S contains all the activated `cand/3` constraints in addition to `start` constraint

A rule in w_p is fired through the "apply" transition. However, the "apply" transition is applicable only to a state with an empty goal. This transition also ensures that the highest priority rule among the set of applicable rules is the one fired [8].

Table 4. Computation of step 3 (Candidate Set Resolution)

$\xmapsto[\text{apply start}]{}$	$\langle\{$`candList([])`$\}, S\rangle$	
$\xmapsto[\text{introduce}]{}$	$\langle\phi, S\rangle$	
$\xmapsto[\text{apply collect}]{}$	$\langle\{$`candList(L)`$\}, S\rangle$	
$\xmapsto[\text{introduce}]{}$	$\langle\phi, S\rangle$	This computation step and the one above are repeated till no more `cand/3` constraints are available in store
$\xmapsto[\text{apply choose}]{}$	$\langle\{$`fire(R,IDs),id(N)`$\}, S\rangle$	
$\xmapsto[\text{introduce}]{}$	$\langle\phi, S\rangle$	The store S at this step contains only one `fire/2` constraint

Table 5. Computation of step 4 (Rule Application)

$\xmapsto[\text{apply apply}]{}$	$\langle G, S\rangle$	The goal G contains the body of the fired rule, and the store S contains the rest of the activated constraints after applying the chosen rule

As shown in the last step of each of table 3 and 4, the result step contains an empty goal. Thus before firing any rule in table 5, the state contains an empty goal. Table 5 shows the rule application step. Therefore, the transformed program only fires the rule when the goal of the state is empty.

In Section 4.1, we showed how the rule matching step finds the set of all applicable rules. The set of rules of Table 3 on the other hand chooses the highest priority rule among this set using the built-in constraint `sort/2`.

The only difference between the goal of S_1 and $S2$ is the set of auxiliary constraints. As shown, the conditions required by w_p to fire a rule in P are the same as the conditions required by w_t to fire a rule in $T(P)$. Consequently, the same rules will be fired in both programs with the same order. Thus, both derivations add the same constraints to the final store. Thus, at the end of both derivations the only difference in the result states ${S_1}'$ and ${S_1}' \cup Aux'$ is auxiliary constraints. Accordingly, omitting the auxiliary constraints from ${S_1}' \cup Aux'$ will result in ${S_1}'$. Hence, we proved that the transformed program $T(P)$ is equivalent to the source program P.

$\square$

6 Conclusion

This paper introduced a source-to-source transformation approach to implement a set of CHR operational semantics that have a different execution model than the refined operational semantics. The source programs written in different operational semantics are transformed into equivalent programs written under the refined operational semantics. Moreover, the execution of the transformed program does not need accessing the compiler or changing the runtime environment.

The transformation approach allows a different rule application choice when there is more than one applicable rule compared to the top-down program order of the refined operational semantics. Moreover, it allows forward and inverse execution of CHR programs. A sketch proof is provided to show the equivalence between the transformed programs and the source programs.

For future work, we intend to extend the transformation approach to implement a larger set of CHR operational semantics by incorporating additional properties to the current model. In addition, we intend to investigate the result of combining the properties of the execution model of different operational semantics such as combining inverse CHR and CHR^{r^p}.

References

1. Abdennadher, S., Schütz, H.: $\text{CHR}^{\vee}$: A Flexible Query Language. In: Andreasen, T., Christiansen, H., Larsen, H.L. (eds.) FQAS 1998. LNCS (LNAI), vol. 1495, pp. 1–14. Springer, Heidelberg (1998)
2. Abdennadher, S., Sharaf, N.: Visualization of CHR through Source-to-Source Transformation. In: Dovier, A., Costa, V.S. (eds.) Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012). Leibniz International Proceedings in Informatics (LIPIcs), vol. 17, Dagstuhl, Germany, pp. 109–118. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2012)
3. Duck, G.J., Stuckey, P.J., Garcia de la Banda, M., Holzbaur, C.: The Refined Operational Semantics of Constraint Handling Rules. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 90–104. Springer, Heidelberg (2004)
4. Frühwirth, T.: Theory and Practice of Constraint Handling Rules, Special Issueon Constraint Logic Programming . The Journal of Logic Programming 37(13), 95–138 (1998)
5. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (August 2009)
6. Frühwirth, T., Di Pierro, A., Wiklicky, H.: Probabilistic ConstraintHandling Rules. In: Comini, M., Falaschi, M. (eds.) WFLP 2002: Proc. 11th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers, vol. 76 (June 2002)
7. Frühwirth, T.W., Holzbaur, C.: Source-to-Source Transformation for a Class of Expressive Rules. In: Buccafurri, F. (eds.) APPIA-GULP-PRODE, pp. 386–397 (2003)
8. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for chr. In: Leuschel, M., Podelski, A. (eds.) PPDP, pp. 25–36. ACM (2007)
9. Sneyers, J., Meert, W., Vennekens, J.: CHRiSM: CHance Rules induce Statistical Models. In: Proceedings of the Sixth International Workshop on Constraint Handling Rules, pp. 62–76 (2009)
10. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As Time Goes By: Constraint Handling Rules - A Survey of CHR Research between 1998 and 2007 10(1), 1–47 (2010)

11. Sneyers, J., Van Weert, P., Schrijvers, T., Demoen, B.: Aggregates in Constraint Handling Rules. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 446–448. Springer, Heidelberg (2007)
12. Van Weert, P., Sneyers, J., Schrijvers, T., Demoen, B.: Extending CHR with Negation as Absence. Technical report CW 452, 125–140 (July 2006)
13. Wielemaker, J., Frühwirth, T., De Koninck, L., Triska, M., Uneson, M.: SWI Prolog Reference Manual 6.2.2. (September 2012)
14. Zaki, A., W. Frühwirth, T., Abdennadher, S.: Towards inverse execution of constraint handling rules. TPLP, 13 (4-5-Online-Supplement) (2013)