Unfolding in CHR

Paolo Tacchella
Dipartimento di Scienze
dell'Informazione
Università di Bologna
Via Mura Anteo Zamboni 7,
40127 Bologna, Italy
Paolo.Tacchella@cs.unibo.it

Maria Chiara Meo
Dipartimento di Scienze
Università di Chieti
Viale Pindaro 42,
65127 Pescara, Italy
cmeo@unich.it

Maurizio Gabbrielli
Dipartimento di Scienze
dell'Informazione
Università di Bologna
Via Mura Anteo Zamboni 7,
40127 Bologna, Italy
gabbri@cs.unibo.it

Abstract

Program transformation is an appealing technique which allows to improve run-time efficiency, space-consumption and more generally to optimize a given program. Essentially it consists of a sequence of syntactic program manipulations which preserves some kind of semantic equivalence. One of the basic operations which is used by most program transformation systems is unfolding which consists in the replacement of a procedure call by its definition. While there is a large body of literature on transformation and unfolding of sequential programs, very few papers have addressed this issue for concurrent languages and, to the best of our knowledge, no one has considered unfolding of CHR programs.

This paper is a first attempt to define a correct unfolding system for CHR programs. We define an unfolding rule, show its correctness and discuss some conditions which can be used to delete an unfolded rule while preserving the program meaning.

Categories and Subject Descriptors D.3.1 [*Programming languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.2 [*Programming Languages*]: Language Classifications - *Constraint and logic languages*; F.3.2 [*Logic and meanings of programs*]: Semantics of Programming Languages

General Terms Languages, Theory.

Keywords Unfolding, Constraint Handling Rules, Program transformation.

1. Introduction

CHR [5, 7] is a committed-choice declarative language which has been specifically designed for writing constraint solvers. There is nowadays a very large literature on CHR, ranging from theoretical aspects to implementations and applications (more than 1000 papers mentioning CHR are reported at the web site http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/). However, only a few papers, notably [6, 9, 11], consider source to source transformation of CHR programs. This is not surprising, since program transformation is in general very difficult for (logic) concurrent languages and in case of CHR it is even more complicated, as we discuss later.

While [6] focuses on specialization of a program for a given goal, here we consider unfolding. This is a basic operation of any source to source transformation (and specialization) system and essentially consists in the replacement of a procedure call by its definition. While this operation can be performed rather easily for sequential languages, and indeed in the field of logic programming it was first investigated by Tamaki and Sato more than twenty years ago [12], when considering logic concurrent languages it becomes quite difficult to define reasonable conditions which ensure its correctness. This is mainly due to the presence of guards in the rules: Intuitively, when unfolding a rule r by using a rule r' (i.e. when replacing in the body of r a "call" of a procedure by its definition r') it could happen that some guard in r' is not satisfied "statically" (i.e. when we perform the unfold), even though it could become satisfied later when the unfolded rule is actually used. If we move the guard of r' in the unfolded version of r we can then loose some computations (because the guard is anticipated). This means that if we want to preserve the meaning of a program we cannot replace the rule r by its unfolded version, and we have to keep both the rules. For CHR the situation is further complicated by the presence of multiple heads in the rules. The problem here is that even though the guards are satisfied, when unfolding a body B of a rule r by using the (multiple) head H of another rule r', we cannot be sure that at run-time all the atoms in H will indeed be used to rewrite B, since in general B could be in a conjunction with other atoms. This technical point, that one can legitimately find obscure now, will be further clarified in Section 5.

Despite these technical problems, the study of unfolding techniques for concurrent languages, and for CHR in particular, is important as it could lead to significant improvements in the efficiency of programs.

In this paper we then define an unfolding rule for CHR programs and show that it preserves the semantics of the program in terms of qualified answers (a notion already defined in the literature). We also provide a syntactic condition which allows to replace in a programs a rule by its unfolded version while preserving qualified answers. Even though the idea of the unfolding is straightforward, its technical development is complicated by the presence of guards and multiple heads, as previously mentioned. In particular, it is not immediate to identify conditions which allow to replace the original rule by its unfolded version. Moreover, a further reason of complication comes from the fact that we consider the reference semantics (called ω_t) defined in [3] which avoids trivial non termination by using a, so called, token store (see next section). Due to the presence of this token store, in order to define correctly the

PPDP'07 July 14–16, 2007, Wroclaw, Poland.

unfolding we have to slightly modify the syntax of CHR programs by adding to each rule a local token store. The resulting programs are called annotated and we define their semantics by providing a (slightly) modified version of the semantics ω_t, which is proven to preserve the qualified answers.

The remaining of this paper is organized as follows. Next section contains some notations used in the paper and the syntax of CHR. The operational semantics of ω_t [3] and of the modified semantics ω_t' are presented in Section 3. Section 4 defines the unfolding rule and prove its correctness. Section 5 discuss the problems related to the replacement of a rule by its unfolded version and gives a correctness condition which holds for a specific class of rules. Finally Section 6 concludes by discussing also some related work.

2. Preliminaries

In this section we introduce the syntax of CHR and some notations and definitions we will need in the following. CHR uses two kinds of constraints: the built-in and the CHR ones, also called user-defined. Built-in constraints are defined by $c ::= d|c \wedge c|\exists_x c$, where d is an atomic formula (or atom). These constraints are handled by an existing solver and we assume given a (first order) theory CT which describes their meaning. We use c, d to denote built-in constraints, h, k, s, p, q to denote CHR constraints and a, b, g, f to denote both built-in and user-defined constraints (we will call these generically constraints). We also denote by `false` any inconsistent (conjunction of) constraints. The capital versions will be used to denote multisets (or sequences) of constraints.

The notation $\exists_{-V}\phi$, where V is a set of variables, denotes the existential closure of a formula ϕ with the exception of the variables in V which remain unquantified. $Fv(\phi)$ denotes the free variables appearing in ϕ. Moreover, if $\bar{t} = t_1, \ldots t_m$ and $\bar{t}' = t'_1, \ldots t'_m$ are sequences of terms then the notation $p(\bar{t}) = p'(\bar{t}')$ represents the set of equalities $t_1 = t'_1, \ldots, t_m = t'_m$ if $p = p'$, and it is undefined otherwise. Analogously, if $H = h_1, \ldots, h_k$ and $H' = h'_1, \ldots, h'_m$ are sequences of constraints, the notation $H = H'$ represents the set of equalities $h_1 = h'_1, \ldots, h_k = h'_k$. Finally, multi-set union is represented by symbol $\uplus$.

2.1 CHR syntax

As shown by the following definition, a *CHR program* consists of a set of rules which can be divided into three types: *simplification*, *propagation* and *simpagation* rules. The first kind of rules is used to rewrite CHR constraints into simpler ones, while second one allows to add new redundant constraints which may cause further simplification. Simpagation rules allow to represent both simplification and propagation rules.

DEFINITION 1. CHR SYNTAX *[5]. A CHR program is a finite set of CHR rules. There are three kinds of CHR rules:*
*A **simplification** rule has the form:*

$$r@H \Leftrightarrow C \mid B$$

*A **propagation** rule has the form:*

$$r@H \Rightarrow C \mid B$$

*A **simpagation** rule has the form:*

$$r@H_1 \setminus H_2 \Leftrightarrow C \mid B,$$

where r is a unique identifier of the rule, H, H_1 and H_2 are sequences of user-defined constraints (called heads), C is a possibly empty multiset of built-in constraints (guard) and B is a possibly empty multiset of (built-in and user-defined) constraints (body). A CHR goal *is a multiset of (both user-defined and built-in) constraints.*

A *simpagation* rule can simulate both simplification and propagation rule by considering, respectively, either H_1 or H_2 empty (with $(H_1, H_2) \neq \emptyset$). In the following we will then consider in the formal treatment only simpagation rules.

When considering unfolding we need to consider a slightly different syntax, where rule identifiers are not necessarily unique, atoms in the body are associated with an identifier and where each rule is associated with a local token store T. More precisely, we define an identified CHR constraint (or identified atom) $h\#i$ as a CHR constraint h, associated with an integer i which allows to distinguish different copies of the same constraint. We will also use the functions *chr(h#i)=h* and *id(h#i)=i*, possibly extended to sets and sequences of identified CHR constraints in the obvious way. Given a goal G, we denote by $\tilde{G}$ one of the possible identified versions of G. $Goals$ is the set of all (possibly identified) goals.

DEFINITION 2. CHR ANNOTATED SYNTAX. *Let us define a token as an object of the form $r@i_1, \ldots, i_l$, where r is the name of a rule and $i_1, \ldots, i_l$ is a sequence of identifiers. A token store is a set of tokens.*
*An **annotated** rule has then the form:*

$$r@H_1 \setminus H_2 \Leftrightarrow C \mid \tilde{B}; T$$

where r is an identifier, H_1 and H_2 are sequences of user-defined constraints, $\tilde{B}$ is a sequence of built-in and identified CHR constraints such that different (occurrences of) CHR constraints have different identifiers, and T is a token store, called the local token store of rule r. An annotated CHR program is a finite set of annotated CHR rules.

Intuitively, identifiers are used to distinguish different occurrences of the same atom in a rule. The identified atoms can be obtained by using a suitable function which associates a (unique) integer to each atom. More precisely, let B be a goal which contains m CHR-constraints. We assume that the function $I_n^{n+m}(B)$ identifies each CHR constraint in B by associating to it a unique integer in $[n+1, m+n]$ according to the lexicographic order.

On the other hand, the token store allows to memorize some tokens, where each token describes which (propagation) rule has been used for reducing which identified atoms. As we discuss in the next section, the use of this information was originally proposed in [1] and then further elaborated in the semantics defined in [3] in order to avoid trivial non termination arising from the repeated application of the same propagation rule to the same constraints. Here we simply incorporate this information in the syntax, since we will need to manipulate it in our unfolding rule.

Given a CHR program P, by using the function $I_n^{n+m}(B)$ and an initially empty local token store we can construct its annotated version as follows.

DEFINITION 3. *Let P be a CHR program. Then its annotated version is defined as follows:*

$$\begin{aligned} Ann(P) = \{ \quad & r@H_1 \setminus H_2 \Leftrightarrow C \mid I_0^m(B); \emptyset \mid \\ & r@H_1 \setminus H_2 \Leftrightarrow C \mid B \in P \text{ and} \\ & m \text{ is the number of CHR-constraints in } B \quad \}. \end{aligned}$$

Notation
In the following examples, given a (possibly annotated) rule

$$r@H_1 \setminus H_2 \Leftrightarrow C \mid B(;T),$$

we write it as

$$r@H_2 \Leftrightarrow C \mid B(;T),$$

if H_1 is empty and we write it as

$$r@H_1 \Rightarrow C \mid B(;T),$$

if H_2 is empty.

That is, we maintain also the notation previously introduced for simplification and propagation rules. Moreover, if $C = \texttt{true}$, then $\texttt{true}\mid$ is omitted. Finally, if in annotated rule the token store is empty we simply omit it.

3. CHR operational semantics

This section introduces the reference semantics ω_t [3], in particular the variant that modifies the token set only after the application of a propagation rule (for the sake of simplicity, we omit indexing the relation with the name of the program).

Afterward we define a slightly different operational semantics, called ω_t', which considers annotated programs and which will be used to prove the correctness of our unfolding rules (via some form of equivalence between ω_t' and ω_t).

We describe the operational semantics ω_t, introduced in [3], by using a transition system

$$T_{\omega_t} = (Conf_t, \longrightarrow_{\omega_t}).$$

Configurations in $Conf_t$ are tuples of the form $\langle G, \tilde{S}, c, T\rangle_n$ with the following meaning. G, the goal, is a multiset of constraints to be evaluated. The CHR constraint store $\tilde{S}$ is the set of identified CHR constraints that can be matched with rules in the program P. The built-in constraint store c is a conjunction of built-in constraints. The *propagation history* T is a set of tokens of the form $r@i_1, \ldots, i_l$, where r is the name of the applied propagation rule and $i_1, \ldots, i_l$ is the sequence of identifiers associated to the constraints to which the head of the rule is applied. This is needed to prevent trivial non-termination for propagation rules. If one does not consider tokens (as in the original semantics of [5]) it is clear from the transition system that if a propagation rule can be applied once it can be applied infinitely many times thus originating an infinite computation (no fairness assumptions are made here). On the other hand, by using tokens one can ensure that a propagation rule is used to reduce a sequence of constraints only if the same rule has not been used before on the same sequence of constraints, thus avoiding trivial infinite computations (arising from the application of the same rule to the same constraints). Finally the counter n represents the next free integer which can be used to number a CHR constraint. As previously mentioned, the first idea of using a token store to avoid trivial non termination was described in [1].

Given a goal G, the *initial configuration* has the form

$$\langle G, \emptyset, \texttt{true}, \emptyset\rangle_1.$$

A *final configuration* has either the form $\langle G', \tilde{S}, \texttt{false}, T\rangle_n$ when it is *failed* or it has the form $\langle \emptyset, \tilde{S}, c, T\rangle_n$ when it represents a successful termination (since there are no more applicable rules).

The relation $\longrightarrow_{\omega_t}$ (of the transition system of the operational semantics ω_t) is defined by the rules in Table 1: the **Solve** rule moves a built-in constraint from goal store to the built-in constraint store; the **Introduce** identifies and moves a CHR (or used defined) constraint from the goal store to the CHR constraint store and the **Apply** rule chooses a program rule r, for which matching between constraints in CHR store and the ones in the head of r exists, it checks that the guard of r is entailed by the built-in constraint store, considering the matching substitution, and it verifies that the token that would be eventually added by $Apply$ in the token store is not already present, than it fires the rule.

3.1 The modified semantics ω_t'

We now define the semantics ω_t' which considers annotated rules. This semantics differs from ω_t in two aspects.

First, in ω_t' the goal store and the CHR store are fused in a unique generic *store*, where CHR constraints are immediately labeled. As a consequence, we do not need anymore the Introduce rule and every CHR constraint in the body of an applied rule is immediately utilizable for rewriting.

The second difference concerns the shape of the rules. In fact, each annotated rule r has a local token store (which can be empty) that is associated to it and which is used to keep trace of the propagation rules that are used to unfold the body of r. Note also that here, differently from the case of the propagation history in ω_t, the token store associated to the real computation can be updated by adding more tokens at once (because an unfolded rule with many token in its local token store has been used).

In order to define formally ω_t' we need a function $inst$ which is defined as follows.

DEFINITION 4. *Let $Token$ be the set of all possible token set and let $\mathbb{N}$ be the set of natural numbers. We denote by $inst : Goals \times \{Token\} \times \mathbb{N} \to Goals \times \{Token\} \times \mathbb{N}$ the function such that $inst(\tilde{B}, T, n) = (\tilde{B}', T', m)$, where*

- *$\tilde{B}$ is an identified CHR goal,*
- *$(\tilde{B}', T')$ is obtained from $(\tilde{B}, T)$ by incrementing each identifier in $(\tilde{B}, T)$ with n and*
- *m is the greatest identifier in $(\tilde{B}', T')$.*

We describe now the operational semantics ω_t' for annotated CHR programs by using, as usual, a transition system

$$T_{\omega_t'} = (Conf_t', \longrightarrow_{\omega_t'}).$$

Configurations in $Conf_t'$ are tuples of the form $\langle \tilde{S}, c, T\rangle_n$ with the following meaning. $\tilde{S}$ is the set of identified CHR constraints that can be matched with rules in the program P and built-in constraints. The built-in constraint store c is a conjunction of built-in constraints and T is a set of tokens, while the counter n represents the last integer which was used to number the CHR constraints in $\tilde{S}$.

Given a goal G, the *initial configuration* has the form

$$\langle I_0^m(G), \texttt{true}, \emptyset\rangle_m,$$

where m is the number of CHR constraints in G. A *final configuration* has either the form $\langle \tilde{S}, \texttt{false}, T\rangle_n$ when it is *failed* or it has the form $\langle \tilde{S}, c, T\rangle_n$ when it represents a successful termination, since there are no more applicable rules.

The relation $\longrightarrow_{\omega_t'}$ (of the transition system of the operational semantics ω_t') is defined by the rules in Table 2. Let us discuss briefly the rules.

Solve' moves a built-in constraint from the store to the built-in constraint store;

Apply' uses the rule $r@H_1' \backslash H_2' \Leftrightarrow D \mid \tilde{B}; T_r$ provided that exists a matching substitution θ such that $chr(\tilde{H}_1, \tilde{H}_2) = (H_1', H_2')\theta$, D is entailed by the built-in constraint store of the computation and $r@id(\tilde{H}_1, \tilde{H}_2) \notin T$; $\tilde{H}_2$ is replaced by $\tilde{B}$, where the identifier are suitably incremented by $inst$ function and $chr(\tilde{H}_1, \tilde{H}_2) = (H_1', H_2')$ is added to built-in constraint store.

In order to show the equivalence of the semantics ω_t and ω_t' we now define the notion of observables that we consider: these are the "qualified answers" (already used in [5]).

DEFINITION 5. (QUALIFIED ANSWERS). *Let P be a CHR program and let G be a goal. The set $\mathcal{QA}_P(G)$ of qualified answers*

Solve$_{\omega_t}$	$\dfrac{CT \models c \wedge C \leftrightarrow C' \text{ and c is a built-in constraint}}{\langle\{c\} \uplus G, \tilde{S}, C, T\rangle_n \longrightarrow_{\omega_t} \langle G, \tilde{S}, C', T\rangle_n}$
Introduce$_{\omega_t}$	$\dfrac{\text{h is a user-defined constraint}}{\langle\{h\} \uplus G, \tilde{S}, C, T\rangle_n \longrightarrow_{\omega_t} \langle G, \{h\#n\} \cup \tilde{S}, C, T\rangle_{n+1}}$
Apply$_{\omega_t}$	$\dfrac{r@H'_1 \setminus H'_2 \Leftrightarrow D \mid B \in P \quad x = Fv(H'_1, H'_2) \quad CT \models C \rightarrow \exists_x((chr(\tilde{H}_1, \tilde{H}_2) = (H'_1, H'_2)) \wedge D)}{\langle G, \{\tilde{H}_1\} \cup \{\tilde{H}_2\} \cup \tilde{S}, C, T\rangle_n \longrightarrow_{\omega_t} \langle B \uplus G, \{\tilde{H}_1\} \cup \tilde{S}, (chr(\tilde{H}_1, \tilde{H}_2) = (H'_1, H'_2)) \wedge C, T'\rangle_n}$
	where $r@id(\tilde{H}_1, \tilde{H}_2) \notin T$ and $T' = T \cup \{r@id(\tilde{H}_1, \tilde{H}_2)\}$ if $\tilde{H}_2 = \emptyset$ otherwise $T' = T$.

Table 1. The transition system T_{ω_t} for the ω_t semantics

Solve'	$\dfrac{CT \models C \wedge c \leftrightarrow C' \text{ and } c \text{ is a built-in constraint}}{\langle\{c\} \cup \tilde{G}, C, T\rangle_n \mapsto_{\omega'_t} \langle\tilde{G}, C', T\rangle_n}$
Apply'	$\dfrac{(r@H'_1 \setminus H'_2 \Leftrightarrow D \mid \tilde{B}; T_r) \in P, \quad x = Fv(H'_1, H'_2) \quad CT \models C \rightarrow \exists_x((chr(\tilde{H}_1, \tilde{H}_2) = H'_1, H'_2) \wedge D)}{\langle\tilde{H}_1 \cup \tilde{H}_2 \cup \tilde{G}, C, T\rangle_n \mapsto_{\omega'_t} \langle\tilde{B}' \cup \tilde{H}_1 \cup \tilde{G}, (chr(\tilde{H}_1, \tilde{H}_2) = (H'_1, H'_2) \wedge C, T'\rangle_m}$
	where $(\tilde{B}', T'_r, m) = inst(\tilde{B}, T_r, n); r@id(\tilde{H}_1, \tilde{H}_2) \notin T$ and $T' = T \cup \{r@id(\tilde{H}_1, \tilde{H}_2)\} \cup T'_r$ if $\tilde{H}_2 = \emptyset$ otherwise $T' = T \cup T'_r$.

Table 2. The transition system $T_{\omega'_t}$ for the ω'_t semantics

for the query G in the program P is defined as follows:

$$\mathcal{QA}_P(G) = \{\exists_{-Fv(G)} K \wedge d \mid \langle G, \emptyset, \texttt{true}, \emptyset\rangle_1 \rightarrow^*_{\omega_t} \langle\emptyset, \tilde{K}, d, T\rangle_n \not\rightarrow_{\omega_t}\} \cup \{\texttt{false} \mid \langle G, \emptyset, \texttt{true}, \emptyset\rangle_1 \rightarrow^*_{\omega_t} \langle G', \tilde{K}, \texttt{false}, T\rangle_n\}.$$

Analogously we can define the qualified answer of an annotated program.

DEFINITION 6. (QUALIFIED ANSWER FOR ANNOTATED PROGRAMS). *Let P be an annotated CHR program and let G be a goal with m CHR constraints. The set $\mathcal{QA'}_P(G)$ of qualified answers for the query G in the annotated program P is defined as follows:*

$$\mathcal{QA'}_P(G) = \{\exists_{-Fv(G)} K \wedge d \mid \langle I_0^m(G), \texttt{true}, \emptyset\rangle_m \rightarrow^*_{\omega'_t} \langle\tilde{K}, d, T\rangle_n \not\rightarrow_{\omega'_t}\} \cup \{\texttt{false} \mid \langle I_0^m(G), \texttt{true}, \emptyset\rangle_m \rightarrow^*_{\omega'_t} \langle\tilde{G'}, \texttt{false}, T\rangle_n\}.$$

The following result shows the equivalence of the two semantics w.r.t. (the equivalence induced by) qualified answers. The proof is easy by definition of ω_t and ω'_t.

PROPOSITION 1. *Let P and $Ann(P)$ be respectively a CHR program and its annotated version. Then, for every goal G,*

$$\mathcal{QA}_P(G) = \mathcal{QA'}_{Ann(P)}(G)$$

holds.

4. The unfolding rule

In this section we define the *unfold operation* for CHR simpagation rules. As a particular case we obtain also unfolding for simplification and propagation rules, as these can be seen as particular cases of the former.

The unfolding allows to replace a conjunction S of constraints (which can be seen as a procedure call) in the body of a rule r by the body of a rule v, provided that the head of v matches with S. More precisely, assume that the head H of v, instantiated by a substitution θ, matches with the conjunction S (in the body of r). Then the unfolded rule is obtained from r by performing the following steps: 1) the new guard in the unfolded rule is the conjunction of the guard of r with the guard of v, the latter instantiated by θ and without those constraints that are entailed by the built-in constraints which are in r; 2) the body of v and the equality $H = S$ are added to the body of r (equality here is interpreted as syntactic equality); 3) the conjunction of constraints S can be removed, partially removed or left in the body of the unfolded rule, depending on the fact that v is a simplification, a simpagation or a propagation rule, respectively; 4) as for the local token store T_r associated to every rule r, this is updated consistently during the unfolding operations in order to avoid that a propagation rule is used twice to unfold the same sequence of constraints.

Before formally defining the unfolding we need to define the function

$$clean : Goals \times Token \rightarrow Token,$$

as follows: $clean(\tilde{B}, T)$ deletes from T all the tokens for which at least one identifier is not present in the identified goal $\tilde{B}$. More formally

$$clean(\tilde{B}, T) = \{t \in T \mid \quad t = r@i_1, \ldots, i_k \text{ and } i_j \in id(\tilde{B}), \text{ for each } j \in [1, k]\}.$$

Recall also that we defined *chr(h#i)=h*.

DEFINITION 7. (UNFOLD). *Let P be an annotated CHR program and let $r, sp \in P$ be two annotated rules such that:*

$$r@H_1 \setminus H_2 \quad \Leftrightarrow \quad D \mid \tilde{K}, \tilde{S}_1, \tilde{S}_2, C; T \text{ and}$$
$$sp@H'_1 \setminus H'_2 \quad \Leftrightarrow \quad D' \mid \tilde{B}; T' \text{ is a simpagation rule}$$

where $chr(\tilde{S}_1, \tilde{S}_2)$ is identical to $(H'_1, H'_2)\theta$, that is, the constraints H'_1 in the head of rule sp match with $chr(\tilde{S}_1)$ and H'_2 matches with $chr(\tilde{S}_2)$ by using the substitution θ. Furthermore assume that C is the conjunction of all the built-in constraints in the

body of r, that m is the greatest identifier which appears in the rule r and that $(\tilde{B}_1, T_1, m_1) = inst(\tilde{B}, T', m)$. Then the unfolded *rule is:*

$$r@H_1 \backslash H_2 \Leftrightarrow D, (D''\theta) \mid \tilde{K}, \tilde{S}_1, \tilde{B}_1, C, chr(\tilde{S}_1, \tilde{S}_2) = (H_1', H_2'); T''$$

where $sp@id(\tilde{S}_1, \tilde{S}_2) \notin T$, $D'' = D' \backslash V$, $V \subseteq D'$, either $CT \models C \rightarrow V\theta$ or $CT \models D \rightarrow V\theta$, the constraint $(D, (D''\theta))$ is satisfiable and

- *if $H_2' = \emptyset$ then $T'' = clean((\tilde{K}, \tilde{S}_1), T) \cup T_1 \cup \{sp@id(\tilde{S}_1)\}$*
- *if $H_2' \neq \emptyset$ then $T'' = clean((\tilde{K}, \tilde{S}_1), T) \cup T_1$.*

Note that we use the function $inst$ (defined in Definition 4) in order to increment the value of the identifiers associated to atoms in the unfolded rule. This allows us to distinguish the new identifiers introduced in the unfolded rule from the old ones. Note also that the condition on the token store is needed to obtain a correct rule. Consider for example a ground annotated program $P = \{r_1@h \Leftrightarrow \tilde{k},\ r_2@k \Rightarrow \tilde{s},\ r_3@s, s \Leftrightarrow \tilde{B}\}$ and let h be the start goal. In this case the unfolding could change the semantics if the token store were not used. In fact, according to the semantics proposed in Table 1 or 2, we have the following computation: $\tilde{h} \rightarrow^{(r_1)} \tilde{k} \rightarrow^{(r_2)} \tilde{k}, \tilde{s} \not\rightarrow_{\omega_t}$. On the other hand, considering an unfolding without the update of the token store one would have $r_1@h \Leftrightarrow \tilde{k} \overset{\text{unfold using } r_2}{\longrightarrow} r_1@h \Leftrightarrow \tilde{k}, \tilde{s} \overset{\text{unfold using } r_2}{\longrightarrow} \underline{r_1@h \Leftrightarrow \tilde{k}, \tilde{s}, \tilde{s}} \overset{\text{unfold using } r_3}{\longrightarrow} r_1@h \Leftrightarrow \tilde{k}, \tilde{B}$

so, starting from the constraint h we could arrive to constraint k, B, that is not possible in the original program (the clause obtained after the wrongly applied unfolding rule is underlined).

As previously mentioned, the unfolding rules for simplification and propagation can be obtained as particular cases of Definition 7, by setting $H_1' = \emptyset$ and $H_2' = \emptyset$, respectively, and by considering accordingly the resulting unfolded rule. In the following examples we will use $\odot$ to denote both $\Leftrightarrow$ and $\Rightarrow$.

EXAMPLE 1. *The following program $P = \{r_1, r_2, \bar{r}_2\}$ deduces information about genealogy. Predicate f is considered as father, g as grandfather, gs as grandson and gg as great-grandfather. The following rules are such that we can unfold some constraints in the body of r_1 using the rule r_2 [$\bar{r}_2$].*

$$\begin{array}{l} r_1@f(X,Y), f(Y,Z), f(Z,W) \odot g(X,Z)\#1, \\ \qquad f(Z,W)\#2, gs(Z,X)\#3. \\ r_2@g(X,Y), f(Y,Z) \odot gg(X,Z)\#1. \\ \bar{r}_2@g(X,Y) \backslash f(Y,Z) \Leftrightarrow gg(X,Z)\#1. \end{array}$$

Now we unfold the body of rule r_1 by using the rule r_2 where we assume $\odot$ $=\!\!\Leftrightarrow$ (so we have a simplification rule). We use $inst(gg(X,Z)\#1, \emptyset, 3) = (gg(X,Z)\#4, \emptyset, 4)$ and a renamed version of r_2

$$r_2@g(X',Y'), f(Y',Z') \Leftrightarrow gg(X',Z')\#1.$$

in order to avoid variable clashes. So the new unfolded rule is:

$$\begin{array}{l} r_1@f(X,Y), f(Y,Z), f(Z,W) \odot gg(X',Z')\#4, \\ \qquad gs(Z,X)\#3, X' = X, Y' = Z, Z' = W. \end{array}$$

Now, we unfold the body of rule r_1 by using the simplification rule $\bar{r}_2$. As before,

$$inst(gg(X,Z)\#1, \emptyset, 3) = (gg(X,Z)\#4, \emptyset, 4)$$

and a renamed version of $\bar{r}_2$

$$\bar{r}_2@g(X',Y') \backslash f(Y',Z') \Leftrightarrow gg(X',Z')\#1.$$

is used to avoid variable clashes. The new unfolded rule is:

$$\begin{array}{l} r_1@f(X,Y), f(Y,Z), f(Z,W) \odot g(X,Z)\#1, \\ \quad gg(X',Z')\#4, gs(Z,X)\#3, X' = X, Y' = Z, Z' = W. \end{array}$$

Finally we unfold the body of r_1 by using the r_2 rule where $\odot$ = $\Rightarrow$ is assumed (so we have a propagation rule). As usual, $inst(gg(X,Z)\#1, \emptyset, 3) = (gg(X,Z)\#4, \emptyset, 4)$ and a renamed version of r_2 is used to avoid variable clashes:

$$r_2@g(X',Y'), f(Y',Z') \Rightarrow gg(X',Z')\#1.$$

and so the new unfolded rule is:

$$\begin{array}{l} r_1@f(X,Y), f(Y,Z), f(Z,W) \odot g(X,Z)\#1, \\ \qquad f(Z,W)\#2, gs(Z,X)\#3, gg(X',Z')\#4, X' = X, \\ \qquad Y' = Z, Z' = W; \{r_2@1, 2\}. \end{array}$$

The following example considers more specialized rules with guards which are not `true`.

EXAMPLE 2. *The following program $P = \{r_1, r_2, \bar{r}_2\}$ specializes the rules introduced in Example 1 to the genealogy of Adam. So here we remember that Adam was father of Seth; Seth was father of Enosh; Enosh was father of Kenan. As before, we consider the predicate f as father, g as grandfather, gs as grandson and gg as great-grandfather.*

$$\begin{array}{l} r_1@f(X,Y), f(Y,Z)f(Z,W) \odot X = Adam, Y = Seth \mid \\ \quad g(X,Z)\#1, f(Z,W)\#2, gs(Z,X)\#3, Z = Enosh. \\ r_2@g(X,Y), f(Y,Z) \odot X = Adam, Y = Enosh \mid \\ \quad gg(X,Z)\#1, Z = Kenan. \\ \bar{r}_2@g(X,Y) \backslash f(Y,Z) \Leftrightarrow X = Adam, Y = Enosh \mid \\ \quad gg(X,Z)\#1, Z = Kenan. \end{array}$$

If we unfold r_1 by using (a suitable renamed version of) r_2, where we assume $\odot$ $=\!\!\Leftrightarrow$, we obtain:

$$\begin{array}{l} r_1@f(X,Y), f(Y,Z)f(Z,W) \odot X = Adam, \\ \quad Y = Seth \mid gg(X',Z')\#4, Z' = Kenan, gs(Z,X)\#3, \\ \quad Z = Enosh, X' = X, Y' = Z, Z' = W. \end{array}$$

When $\bar{r}_2$ is considered to unfold r_1 we have

$$\begin{array}{l} r_1@f(X,Y), f(Y,Z)f(Z,W) \odot X = Adam, \\ \quad Y = Seth \mid g(X,Z)\#1, gg(X',Z')\#4, Z' = Kenan, \\ \quad gs(Z,X)\#3, Z = Enosh, X' = X, Y' = Z, Z' = W. \end{array}$$

Finally if we assume $\odot$ $=\!\!\Rightarrow$ in r_2 from the unfolding we obtain

$$\begin{array}{l} r_1@f(X,Y), f(Y,Z), f(Z,W) \odot X = Adam, \\ \quad Y = Seth \mid g(X,Z)\#1, f(Z,W)\#2, gs(Z,X)\#3, \\ \quad gg(X',Z')\#4, Z' = Kenan, Z = Enosh, X' = X, \\ \quad Y' = Z, Z' = W; \{r_2@1, 2\}. \end{array}$$

Note that $X' = Adam, Y' = Enosh$ are not added to the guard of the unfolded rule because $X' = Adam$ is entailed by the guard of r_1 and $Y' = Enosh$ is entailed by the built-in constraints in the body of r_1.

We prove now the correctness of our unfolding rule. The proof of the following proposition is done by induction on the length of the computations.

PROPOSITION 2. *Let P be an annotated CHR program with $r, v \in P$. Let r' be the result of the unfolding of r w.r.t. v and let P' be the program obtained from P by adding rule r'. Then, for every goal G, $\mathcal{QA}'_{P'}(G) = \mathcal{QA}'_{P}(G)$ holds.*

PROOF[SKETCH]. We prove that $\mathcal{QA}'_{P'}(G) \subseteq \mathcal{QA}'_{P}(G)$. The proof of the other inclusion is an obvious consequence of the operational semantics of CHR, since in a computation step one may apply any applicable rule.

First we recall the definition of rule redundancy, given in [2] in terms of finite computation, by considering annotated CHR programs and the transition system $\rightarrow_{\omega_t'}$. A rule r' is redundant in an

annotated CHR program P' if and only if for all the configurations S

$$\text{If } S \rightarrow^*_{\omega'_t} S_1 \text{ in } P' \text{ then } S \rightarrow^*_{\omega'_t} S_2 \text{ in } P' \setminus \{r'\},$$

where S_1 and S_2 are final configurations and S_1 and S_2 are identical up to renaming of logical variables not in S and logical equivalence of built-in constraints.

Now, by definition of unfolding, we can prove that if r' is the result of the unfolding of r w.r.t. v, with $r, v \in P$ and P' is the program obtained from P by adding rule r', then r' is redundant in P'. In fact, each transition step obtained in P' by using the clause r' can be obtained in P also only using rules r, v and some *solve* transition steps. Therefore for all the configurations S

$$\text{If } S \rightarrow^*_{\omega'_t} S_1 \text{ in } P' \text{ then } S \rightarrow^*_{\omega'_t} S_2 \text{ in } P,$$

where S_1 and S_2 are defined as before. Therefore, by definition of $\mathcal{QA}'$, $\mathcal{QA}'_{P'}(G) \subseteq \mathcal{QA}'_P(G)$ and then the thesis holds.
□

The proof of the following result follows immediately from previous proposition and Proposition 1

COROLLARY 1 (Correctness). *Let P be CHR program and let $Ann(P)$ be its annotated version (as previously defined). Let P' be the program obtained from $Ann(P)$ by adding a rule which is obtained by unfolding a rule in $Ann(P)$. Then, for every G, $\mathcal{QA}'_{P'}(G) = \mathcal{QA}_P(G)$ holds.*

5. Safe rule replacement

Previous corollary shows that we can safely add to a program P a rule resulting from the unfolding, while preserving the semantics of P (in terms of qualified answers). However, when a rule r in program P has been unfolded producing the new rule r', in some cases we would like also to replace r by r' in P, since this could improve the efficiency of the resulting program. Performing such a replacement while preserving the semantics is in general a very difficult task for two reasons.

First of all, anticipating the guard of v in the guard of r (as we do in the unfold operation) could lead to loose some computations when the unfolded rule r' is used rather than the original rule r. This is shown by the following example.

EXAMPLE 3. *Consider the rules*

$$h(X) \Leftrightarrow p(X), q(X).$$
$$p(X) \Leftrightarrow X = a \,|\, .$$
$$q(X) \Leftrightarrow X = a.$$

where we do not consider the identifiers (and the local token store) in the body of rules, because we do not have propagation rules in P.

It is clear that the goal $h(X)$ has a successful derivation which computes $X = a$ by using the above rules. On the other hand, this is not the case if one uses the unfolded rule

$$h(X) \Leftrightarrow X = a \,|\, q(X).$$

together with the rules

$$p(X) \Leftrightarrow X = a \,|\, .$$
$$q(X) \Leftrightarrow X = a.$$

For this reason deleting the unfolded rule in general is not safe.

The second problem is related to multiple heads. In fact, the unfolding that we have defined assume that the head of a rule matches completely with the body of another one, while in general, during a CHR computation, a rule can match with constraints produced by more than one rule and/or introduced by the initial goal. The following example illustrates this point.

EXAMPLE 4. *Let us consider the program*

$$P = \{ \quad r@p(Y) \Leftrightarrow q(Y), h(b).$$
$$\qquad r'@q(Z), h(V) \Leftrightarrow Z = V.\}$$

where we do not consider the identifiers and the token store in the body of rules, because we do not have propagation rules in P.

The unfolding of r by using r' returns the new rule

$$r@p(Y) \Leftrightarrow Y = Z, V = b, Z = V.$$

Now the the program

$$P' = \{ \quad r@p(Y) \Leftrightarrow Y = Z, V = b, Z = V.$$
$$\qquad r'@q(Z), h(V) \Leftrightarrow Z = V.\}$$

where we substitute the original rule by its unfolded version is not semantically equivalent to P. In fact, given the goal $G = p(X), h(a), q(b)$, we have that $(X = a) \in \mathcal{QA}'_P(G)$ ($X = a$ is a qualified answer for G in P) while $(X = a) \notin \mathcal{QA}'_{P'}(G)$.

We have individuated a case in which we can safely replace the original rule r by its unfolded version while maintaining the qualified answers semantics. Intuitively, this holds when: 1) the constraints of the body of r can be rewritten only by CHR rules with a single-head and 2) there exists no rule v which has a multiple head H such that a part of H can match with a part of the constraints introduced in the body of r (that is, there exists no rule v which can be fired by using a part of constraints introduced in the body of r plus some other constraints).

Before defining formally these conditions we need some further notations. First of all, given a rule $r@H_1 \backslash H_2 \Leftrightarrow D \,|\, \tilde{A}; T$, we define two sets. The first one contains a set of pairs, whose first component is a rule that can be used to unfold $r@H_1 \backslash H_2 \Leftrightarrow D \,|\, \tilde{A}; T$, while the second one is the sequence of the identifiers of the atoms in the body of r, which are used in the unfolding.

The second set contains all the rules that can be used for the *partial unfolding* of $r@H_1 \backslash H_2 \Leftrightarrow D \,|\, \tilde{A}; T$, namely is the set of rules that can fire by using at least an atom in the body $\tilde{A}$ of the rule and some others CHR and built-in constraints.

DEFINITION 8. *Let P be an annotated CHR program and let*

$$r@H_1 \backslash H_2 \quad \Leftrightarrow \quad D \,|\, \tilde{A}; T \text{ and}$$
$$r'@H'_1 \backslash H'_2 \quad \Leftrightarrow \quad D' \,|\, \tilde{B}; T'$$

be two annotated rules, such that $r, r' \in P$ and r' is renamed apart w.r.t. r. We define U^+ and $U^\#$ as follows:

- $(r'@H'_1 \backslash H'_2 \Leftrightarrow D' \,|\, \tilde{B}; T', (i_1, \ldots, i_n)) \in U^+_P(r@H_1 \backslash H_2 \Leftrightarrow D \,|\, \tilde{A}; T)$ *if and only if $r@H_1 \backslash H_2 \Leftrightarrow D \,|\, \tilde{A}; T$ can be unfolded with $r'@H'_1 \backslash H'_2 \Leftrightarrow D' \,|\, \tilde{B}; T'$ (by Definition 7) by using the sequence of the identified atoms in $\tilde{A}$ with identifiers $(i_1, \ldots, i_n)$.*
- $r'@H'_1 \backslash H'_2 \Leftrightarrow D' \,|\, \tilde{B}; T' \in U^\#_P(r@H_1 \backslash H_2 \Leftrightarrow D \,|\, \tilde{A}; T)$ *if and only if one of the following holds:*

  - either there exist $\tilde{A}' = (\tilde{A}_1, \tilde{A}_2) \subseteq \tilde{A}$ and a built in constraint C' such that $Fv(C') \cap Fv(r') = \emptyset$, the constraint $D \wedge C'$ is satisfiable, $CT \models (D \wedge C') \rightarrow \exists_x((chr(\tilde{A}_1, \tilde{A}_2) = (H'_1, H'_2)) \wedge D')$, $r'@id(\tilde{A}_1, \tilde{A}_2) \notin T$ and $(r'@H'_1 \backslash H'_2 \Leftrightarrow D' \,|\, \tilde{B}; T', id(\tilde{A}_1, \tilde{A}_2) \notin U^+_P(r@H_1 \backslash H_2 \Leftrightarrow D \,|\, \tilde{A}; T)$

  - or there exist $\tilde{A}' \subseteq \tilde{A}$, a multiset of CHR constraints $H' \neq \emptyset$ and a built in constraint C' such that $\tilde{A}' \neq \emptyset$, $Fv(C') \cap Fv(r') = \emptyset$, the constraint $D \wedge C'$ is satisfiable, $\{chr(A'), H'\} = \{K_1, K_2\}$ and $CT \models (D \wedge C') \rightarrow \exists_x(((K_1, K_2) = (H'_1, H'_2)) \wedge D')$.

Note that if $U_P^+(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)$ contains a pair, whose first component is not a rule with a single atom in the head, then by definition, $U_P^\#(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T) \neq \emptyset$.

Finally, given an annotated CHR program P and an annotated rule $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T$, we define

$$Unf_P(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)$$

as the set of all annotated rules obtained by unfolding the rule $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T$ with a rule in P, by using Definition 7.

We can now give the central definition of this section.

DEFINITION 9. (SAFE RULE REPLACEMENT) *Let P be an annotated CHR program and let $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T \in P$, such that the following holds*

i) $U_P^\#((r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T) = \emptyset$ *and*
ii) $\{U_P^+(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T) \neq \emptyset$ *and for each*

$$r@H_1\backslash H_2 \Leftrightarrow D' \mid \tilde{A}'; T' \in Unf_P(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)$$

we have that $CT \models D \leftrightarrow D'$.

Then we say that the rule $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T$ can be safely replaced (by its unfolded version) in P.

Some explanations are in order here. The condition $CT \models D \leftrightarrow D'$ avoids the problems discussed in Example 3, thus allows the anticipation of the guard in the unfolded rule.

Condition **i**) of previous definition implies that $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T$ can be safely deleted from P only if $U_P^+(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)$ contains only pairs, whose first component is a rule with a single atom in the head. The condition **i)** states that a sequence of identified atoms of body of the rule r can be used to (partially) fire a rule r' only if r can be unfolded with r' by using the same sequence of the identified atoms.

Condition **ii)** states that each annotated clause obtained by the unfolding of r in P must have guard equivalent to that of r. If such a condition is not verified we could have the problem exemplified by the following example.

EXAMPLE 5. *Let us consider the program*

$$P = \{ \quad r@p(Y) \Leftrightarrow q(Y). \\ \quad\quad r'@q(Z) \Leftrightarrow Z = a \mid . \}$$

where we do not consider the identifiers (and the local token store) in the body of rules, because we do not have propagation rules in P.

The unfolding of r by using the rule r' returns the new rule $r@p(Y) \Leftrightarrow Y = a \mid Y = Z$. The program

$$P' = \{ \quad r@p(Y) \Leftrightarrow Y = a \mid Y = Z. \\ \quad\quad r'@q(Z) \Leftrightarrow Z = a \mid . \}$$

is not semantically equivalent to P in terms of qualified answers. In fact, given the goal $G = p(X)$ we have $q(X) \in \mathcal{QA}'_P(G)$, while $q(X) \notin \mathcal{QA}'_{P'}(G)$.

We can now provide the result which shows the correctness of the safe rule replacement condition.

THEOREM 1. *Let P be an annotated program, $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T$ be a rule in P such that $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T$ can be safely replaced in P according to Definition 9. Assume also that*

$$P' \quad = \quad (P \setminus \{(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)\}) \cup \\ \quad\quad\quad Unf_P(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T).$$

Then $\mathcal{QA}'_{P'}(G) = \mathcal{QA}'_P(G)$ for any arbitrary goal G.

PROOF.[SKETCH] First observe that, by Proposition 1, the addition of redundant rules preserves the qualified answers and therefore if $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T$ is a rule in P

$$\mathcal{QA}'_P(G) = \mathcal{QA}'_{P \cup Unf_P(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)}(G)$$

for any arbitrary goal G.

Moreover, by definition of redundant rule, is obvious that the removal of a redundant rule preserves the qualified answers. Then the proof follows by observing that, by Definition 9 (Safe rule replacement), we have that a rule $r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T \in P$ is redundant in $P' \cup Unf_P(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)$.
□

Of course, previous result can be applied to a sequence of program transformations. Let us define such a sequence as follows.

DEFINITION 10. *Let P be an annotated CHR program. An U-sequence of programs starting from P is a sequence of annotated CHR programs $P_0, \ldots, P_n$, such that*

$$P_0 \quad = \quad P \text{ and} \\ P_{i+1} \quad = \quad P_i \setminus \{(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)\}) \cup \\ \quad\quad\quad Unf_{P_i}(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T),$$

where $i \in [0, n-1]$, $(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T) \in P_i$ and $(r@H_1\backslash H_2 \Leftrightarrow D \mid \tilde{A}; T)$ is safety deleting from P_i

Then from Theorem 1 and Proposition 1 we have immediately the following.

COROLLARY 2. *Let P be a program and let $P_0, \ldots, P_n$ be an U-sequence starting from $Ann(P)$. Then $\mathcal{QA}_P(G) = \mathcal{QA}'_{P_n}(G)$ for any arbitrary goal G.*

It is also possible to prove that our unfolding preserves confluence.

The confluence property guarantees that any computation for a goal results in the same final state, no matter which of the applicable rules are applied (see [2] for a formal definition).

PROPOSITION 3 (Confluence). *Let P be a program and let $P_0, \ldots, P_n$ be an U-sequence starting from $Ann(P)$. If P satisfies confluence then P_n satisfies confluence too.*

PROOF. [SKETCH] The proof follows immediately from Corollary 2.

□

6. Conclusions

In this paper we have defined an unfold operation for CHR which preserves the qualified answers of a program. This has been obtained by transforming a CHR program into an annotated one which is then unfolded. The equivalence of the unfolded program and the original (non annotated) one is proved (Corollary 1) by using a slightly modified operational semantics for annotated programs (defined in Section 3). We have then provided a condition that could be used to safely replace a rule by its unfolded version, while preserving qualified answers, for a restricted class of rules.

There are only few other papers that consider source to source transformation of CHR programs. [6] rather than considering a generic transformation system focuses on the specialization of rules w.r.t. a specific goal, analogously to what happen in partial evaluation. In [9] CHR rules are transformed in a relational normal form over which a source to source transformation is performed, however the correctness of such a transformation is not proved. Some form of transformation for probabilistic CHR is considered in [8], while guard optimization was studied in [11].

Both general and goal specific approaches are important in order to define practical transformation systems for CHR: In fact, on one hand one need surely some general unfold rule, on the other hand, given the difficulties in removing rules from the transformed program, some goal specific techniques can help to improve the efficiency of the transformed program for specific classes of goals. A method for deleting redundant CHR rules is considered in [2], however this is based on a semantic check and it is not clear whether it can be transformed in some syntactic program transformation rule.

When considering more generally the field of concurrent logic languages we find a few papers which address the issue of programs transformation. Notable examples are [4] that deals with transformation of concurrent constraint programming (ccp) and [13] that considers Guarded Horn Clauses (GHC). The results in these are not directly applicable to CHR because neither ccp not GHC allow rules with multiple heads.

Our paper can be considered as a first step in the direction of defining a transformation system for CHR programs based on unfolding. This step can be extended along several directions. First of all, the condition that we have provided for safely replacing a rule could be generalized to include more cases. Also, we could extend to CHR some of the other transformations, notably folding, which have been defined in [4] for ccp. Finally we would like to investigate from a practical perspective to what extend the program transformation can improve the performances of the CHR solver. Clearly the application of an unfolded rule avoid some computation steps (assuming that unfolding is done at compile time, of course), even though the increase in the number of rules could vanish this improvement when the original rule cannot be removed. Here it would probably be important to consider some unfolding strategy, in order to decide which rules have to be unfolded.

Acknowledgments

This work was partially developed while the first author was visiting the "Institut für Programmiermethodik und Compilerbau" of the Ulm University. We thank Thom Frühwirth, the CHR team at Ulm and the CHR group of the University of Leuven for many interesting feedbacks and suggestions.

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In G. Smolka, editor, *Third Int'l Conf. on Principles and Practice of Constraint Programming (CP 97)*, Lecture Notes in Computer Science 1330. Springer-Verlag, 1997.

[2] S. Abdennadher and T. Früwirth. Integration and optimization of rule-based constraint solvers. In M. B. (Ed.), editor, *LOPSTR 2003*, LNCS, pages 198–231, Berlin Heidelberg, 2004. Springer-Verlag.

[3] G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP*, pages 90–104, September 2004.

[4] S. Etalle, M. Gabbrielli, and M. C. Meo. Transformations of ccp programs. *ACM Trans. Program. Lang. Syst.*, 23(3):304–395, 2001.

[5] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 20(19):1–679, 1998.

[6] T. Frühwirth. Specialization of concurrent guarded multi-set transformation rules. In S. Etalle, editor, *Logic Based Program Synthesis and Transformation*, Lecture Notes in Computer Science, pages 133 – 148, Verona, August 26 – 28 2004. 14th International Symposium, LOPSTR.

[7] T. Frühwirth, and S. Abdennadher. *Essentials of Constraint Programming*. Springer, April 2003.

[8] T. Frhwirth, A. Di Pierro and, H. Wiklicky. Probabilistic Constraint Handling Rules. *11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)* Selected Papers, Marco Comini and Moreno Falaschi, Eds., Vol. 76 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2002.

[9] T. Frühwirth, and C. Holzbaur. Source-to-Source Tranformation for a Class of Expressive Rules. *APPIA-GULP-PRODE 2003 (AGP 2003)*, Reggio Calabria, Italy, September 2003.

[10] C. Holzbaur, M. J. Garcia de la Banda, D. Jeffery, and P. J. Stuckey. Optimizing Compilation of Constraint Handling Rules. In Proceedings of the 17th International Conference on Logic Programming, pages 74 – 89, LNCS 2237, Springer Verlag, 2001.

[11] J. Sneyers, T. Schrijvers, and B. Demoen. Guard and continuation optimization for occurrence representations of CHR. *CW 420*, K.U. Leuven, 2005.

[12] H. Tamaki and T. Sato. Unfold/Fold transformations of logic programs. In *Proceedings of International Conference on Logic Programming*, pages 127–138, 1984.

[13] K. Ueda and K. Furukawa. Transformation rules for GHC programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988 (FGCS'88)*, pages 582–591, 1988.