Information and Computation 228–229 (2013) 62–82

Contents lists available at SciVerse ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

The expressive power of CHR with priorities

Maurizio Gabbrielli [a], Jacopo Mauro [a,*], Maria Chiara Meo [b]

[a] Department of Computer Science and Engineering/Lab. Focus INRIA, University of Bologna, Via Mura Anteo Zamboni 7, 40127 Bologna, Italy
[b] Department of Economic Studies, University of Chieti-Pescara, Viale Pindaro 42, 65127 Pescara, Italy

ARTICLE INFO

Article history:
Received 16 March 2011
Revised 18 February 2013
Available online 9 May 2013

Keywords:
Expressive power
Language encoding
Constraints

ABSTRACT

Constraint Handling Rules (CHR) is a committed-choice declarative language which has been originally designed for writing constraint solvers and which is nowadays a general purpose language.
Recently the language has been extended by introducing user-definable (static or dynamic) rule priorities. The resulting language allows a better control over execution while retaining a declarative and flexible style of programming.
In this paper we study the expressive power of this language. We first show that, in the presence of priorities, differently from the case of standard CHR, considering more than two atoms in the heads of rules does not augment the expressive power of the language. Next we show that also dynamic priorities do not augment the expressive power w.r.t. static priorities. These results are proved by providing explicitly a translation of one language into another one, which preserves a reference semantics.
Finally we show that CHR with priorities is strictly more expressive than standard CHR (under the theoretical operational semantics). This result is obtained by adapting to the CHR case a notion of language encoding which allows to compare Turing powerful languages.

1. Introduction

Constraint Handling Rules (CHR) [7] is a committed-choice declarative language which has been originally designed for writing constraint solvers and which is nowadays a general purpose language. A CHR program consists of a set of multi-headed guarded (simplification, propagation and simpagation) rules which allow one to rewrite constraints into simpler ones until a solved form is reached. The language is parametric w.r.t. an underlying constraint theory $\mathcal{CT}$ which defines the meaning of basic built-in constraints.

The original theoretical operational semantics for CHR, denoted by ω_t, is non-deterministic, as usual for rule based and concurrent languages. Such non-determinism has to be resolved in the implementations by choosing a suitable execution strategy. Most implementations like SWI Prolog, Yap Prolog, and K.U. Leuven JCHR indeed use the, so-called, refined operational semantics [6] called ω_r which fixes most of the execution strategy. This semantics, unlike the theoretical one, offers a good control over execution, however it is quite low-level and lacks flexibility.

For this reason De Koninck et al. [3] proposed an extension of CHR, called *CHR^{rp}*, for supporting a high-level, explicit form of execution control which is more flexible and declarative than the one offered by the ω_r semantics. This is obtained by introducing explicitly in the syntax of the language rule annotations which allow one to specify the priority of each rule. The operational semantics, in the following denoted by ω_p, is changed accordingly: Rules with higher priority are chosen

* Corresponding author.
E-mail addresses: gabbri@cs.unibo.it (M. Gabbrielli), jmauro@cs.unibo.it (J. Mauro), cmeo@unich.it (M.C. Meo).

0890-5401/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ic.2013.05.001

first. Priorities can be either static, when the annotations are completely defined at compile time, or dynamic, when the annotations contain variables which are instantiated at run-time. We denote with *static CHR^rp^* the sublanguage of *CHR^rp^* obtained by allowing static priorities only.

Even though Sneyers et al. in [14] have shown that any algorithm can be implemented in CHR preserving time and space complexity, yet De Koninck et al. in [3] claimed that "priorities do improve the expressivity of CHR".

In this paper we provide a formal ground for this informal claim by using a notion of expressivity coming from the field of concurrency theory to prove several expressivity results relating CHR, CHR^{rp} and *static* CHR^{rp}. In fact, in this field the issue of the expressive power of a language has received considerable attention in the last years and several techniques and formalisms have been proposed for separating the expressive power of different languages which are Turing powerful (and therefore cannot be properly compared by using the standard tools of computability theory). Such a separation is meaningful both from a theoretical and a pragmatic point of view, since different (Turing complete) languages can provide quite different tools for implementing algorithms. Indeed, some existing techniques for comparing the expressive power of two languages take into account the translation process, trying to formalize how difficult such a process is.

One of these techniques, that we use in this paper, is based on the notion of language encoding, first formalized by De Boer et al. in [2,12,15][1] and can be described as follows. Intuitively, a language $\mathcal{L}$ is more expressive than a language $\mathcal{L}'$ or, equivalently, $\mathcal{L}'$ can be encoded in $\mathcal{L}$, if each program written in $\mathcal{L}'$ can be translated into an $\mathcal{L}$ program in such a way that: (1) the intended observable behavior of the original program is preserved, under some suitable decoding; (2) the translation process satisfies some additional restrictions which indicate how easy this process is and how reasonable the decoding of the observables is. For example, typically one requires that the translation is compositional w.r.t. (some of) the syntactic operators of the language (see for example De Boer et al. [2]).

More precisely, in this paper we use the notion of acceptable encoding, defined in the next section, which imposes the following requirements on the translation. First, similarly to the previous cases, we require that the translation of the goal (in the original program) and the decoding of the results (in the translated program) are homomorphic w.r.t. the conjunction of atoms. This assumption essentially means that our encoding and decoding functions respect the structure of the original goal and of the results (recall that for CHR programs these are constraints, that is, conjunction of atoms). Next we assume that the results to be preserved are the, so-called, qualified answers. Also this is a rather natural assumption, since these are the typical CHR observables for many CHR reference semantics.

To simplify the treatment we assume that both the source and the target language use the same built-in constraints, semantically described by a theory $\mathcal{CT}$, which is not changed in the translation process. It is worth noticing on the other hand that we do not impose any restriction on the program translation.

Our first result shows that, in the presence of static priorities, allowing two or more atoms in the head of rules does not change the expressive power of the language. This result is obtained by providing an acceptable encoding of *static* CHR^{rp} into *static* CHR_2^{rp}, where the latter notation indicates the *static* CHR^{rp} language where at most two atoms are allowed in the heads of rules.

We also show that when considering a slightly different notion of answers, namely data sufficient answers, there exists an acceptable encoding from *static* CHR^{rp} to *static* CHR_2^{rp} even if we add also the requirement that the goal encoding and output decoding functions are the identity. It is worth noting that such a result does not hold for CHR without priorities, as shown by Di Giusto et al. in [4].

Next we prove that dynamic priorities do no augment the expressive power of the language w.r.t. static priorities. This result is obtained by providing an acceptable encoding of CHR^{rp} (with dynamic priorities) into *static* CHR^{rp}.

Finally, we prove a separation result showing that (static) priorities augment the expressive power of CHR, that is CHR^{rp} is strictly more expressive than CHR, in the sense that there exists no acceptable encoding of CHR^{rp} into CHR (with the ω_t semantics).

The remainder of the paper is organized as follows. The next section introduces the languages under consideration with the related semantics and some preliminary notions on language encodings. In Section 3.1 we prove that *static* CHR^{rp} can be encoded in *static* CHR_2^{rp}. In Section 3.2 we provide the translation of dynamic priorities into static ones. In Section 4 we prove the separation result while Section 5 concludes by discussing some related works.

A preliminary version of this paper appeared in [10], however that paper considered also a different semantics (the refined one) and did not contain some of the results presented here. In particular Theorems 1 and 3 are new.

2. Syntax and semantics

In this section we give an overview of CHR syntax with its operational semantics following [7,6] and [3].

2.1. Syntax of CHR

We first need to distinguish the constraints handled by an existing solver, called built-in (or predefined) constraints, from those defined by the CHR program, called user-defined (or CHR) constraints. Therefore we assume a signature Σ on which program terms are defined and two disjoint sets of predicate symbols Π_b for built-in and Π_u for user-defined constraints.

[1] The original terminology of these papers was "language embedding".

reflexivity	$\mathtt{leq}(X,Y) \Longleftrightarrow X = Y \mid true$
antisymmetry	$\mathtt{leq}(X,Y), \mathtt{leq}(Y,X) \Longleftrightarrow X = Y$
transitivity	$\mathtt{leq}(X,Y), \mathtt{leq}(Y,Z) \Longrightarrow \mathtt{leq}(X,Z)$

Fig. 1. A program for defining $\leqslant$ in CHR.

Definition 1 *(Built-in constraint).* A *built-in constraint* $p(t_1,\dots,t_n)$ is an atomic predicate where p is a predicate symbol from Π_b and $t_1,\dots,t_n$ are terms over the signature Σ.

For built-in constraints we assume a (first order) theory $\mathcal{CT}$ which describes their meaning.

Definition 2 *(User-defined constraint).* A *user-defined* (*or CHR*) *constraint* $p(t_1,\dots,t_n)$ is an atomic predicate where p is a predicate symbol from Π_u and $t_1,\dots,t_n$ are terms over the signature Σ.

We use c, d to denote built-in constraints, h, k to denote CHR constraints and a, b, f, g to denote both built-in and user-defined constraints (we call these generally constraints). The capital versions of these notations will be used to denote multiset of constraints. We also denote by *false* any inconsistent conjunction of constraints and with *true* the empty multiset of built-in constraints.

We use "," rather than $\wedge$ to denote conjunction and we often consider a conjunction of atomic constraints as a multiset of atomic constraints. We prefer to use multisets rather than sequences (as in the original CHR papers) because our results do not depend on the order of atoms in the rules. In particular, we use this notation based on multisets in the syntax of CHR.

The notation $\exists_V \phi$, where V is a set of variables, denotes the existential closure of a formula ϕ w.r.t. the variables in V, while the notation $\exists_{-V} \phi$ 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. Finally, we denote by $\bar{t}$ and $\bar{X}$ a sequence of terms and of distinct variables, respectively.

In the following, if $\bar{t} = t_1,\dots,t_m$ and $\bar{t}' = t'_1,\dots,t'_m$ are sequences of terms then the notation $p(\bar{t}) = p'(\bar{t}')$ represents the set of equalities $t_1 = t'_1,\dots,t_m = t'_m$ if $p = p'$, and it is false otherwise. This notation is extended in the expected way to multiset of constraints. Moreover we use $++$ to denote sequence concatenation and $\uplus$ for multiset union.

We follow the logic programming tradition and indicate the application of a substitution σ to a syntactic object t by σt.

To distinguish between different occurrences of syntactically equal constraints, a CHR constraint can be labeled by a unique identifier. The resulting syntactic object is called identified CHR constraint and is denoted by $k\#i$, where k is a CHR constraint and i is the identifier. We also use the functions defined as $\mathtt{chr}(k\#i) = k$ and $\mathtt{id}(k\#i) = i$, possibly extended to sets and sequences of identified CHR constraints in the obvious way, to obtain a set or a sequence of identifiers.

2.2. CHR program

A CHR program is defined as a sequence of three kinds of rules: simplification, propagation and simpagation rules. Intuitively, simplification rewrites constraints into simpler ones, propagation adds new constraints which are logically redundant but may trigger further simplifications, simpagation combines in one rule the effects of both propagation and simplification rules. For simplicity we consider simplification and propagation rules as special cases of a simpagation rule. The general form of a simpagation rule is:

$$r\,@H^k \backslash H^h \Longleftrightarrow D \mid B$$

where r is a unique identifier of a rule, H^k and H^h (the heads) are multisets of CHR constraints, D (the guard) is a possibly empty multiset of built-in constraints and B is a possibly empty multiset of (built-in and user-defined) constraints. If H^k is empty then the rule is a simplification rule. If H^h is empty then the rule is a propagation rule. At least one of H^k and H^h must be non-empty.

In the following when the guard D is empty or *true* we omit $D|$. Also the names of rules are omitted when not needed. For a simplification rule we omit $H^k\backslash$ while we write a propagation rule as $H^k \Rightarrow D \mid B$. A CHR *goal* is a multiset of (both user-defined and built-in) constraints. An example of a CHR program is shown in Fig. 1. This program implements the less or equal predicate, assuming that we have only the equality predicate in the available built-in constraints. The first rule, a simplification, deletes the constraint *leq*(X,Y) if $X = Y$. Analogously the second rule deletes the constraints *leq*(X,Y) and *leq*(Y,X) adding the built-in constraint $X = Y$. The third rule of the program is a propagation rule and it is used to add a constraint *leq*(X,Z) when the two constraints *leq*(X,Y) and *leq*(Y,Z) are found.

2.3. Traditional operational semantics

The theoretical operational semantics of CHR, denoted by ω_t, is given in [6] as a state transition system $T = (Conf, \overset{\omega_t}{\rightarrow}_P)$: Configurations in *Conf* are tuples of the form $\langle G, S, B, T\rangle_n$, where G is the goal (a multiset of constraints that remain to be solved), S is the CHR store (a set of identified CHR constraints), B is the built-in store (a conjunction of built-in

Table 1
Transition rules of ω_t.

Solve	$\langle(c,G),S,C,T\rangle_n \overset{\omega_t}{\rightarrow}_P \langle G,S,c\wedge C,T\rangle_n$ where c is a built-in constraint
Introduce	$\langle(k,G),S,C,T\rangle_n \overset{\omega_t}{\rightarrow}_P \langle G,\{k\#n\}\cup S,C,T\rangle_{n+1}$ where k is a CHR constraint
Apply	$\langle G,H_1\cup H_2\cup S,C,T\rangle_n \overset{\omega_t}{\rightarrow}_P \langle (B,G),H_1\cup S,\theta\wedge D\wedge C,T\cup\{t\}\rangle_n$ where P contains a (renamed apart) rule $r\ @H'_1\backslash H'_2 \Longleftrightarrow D \mid B$ and there exists a matching substitution θ s.t. $\mathtt{chr}(H_1)=\theta H'_1$, $\mathtt{chr}(H_2)=\theta H'_2$, $\mathcal{CT}\models C\rightarrow \exists_{-Fv(C)}(\theta\wedge D)$ and $t=\mathtt{id}(H_1) +\!\!+\ \mathtt{id}(H_2) +\!\!+\ [r]\notin T$

constraints), T is the propagation history (a set of sequences of identifiers used to store the rule instances that have fired) and n is the next free identifier (it is used to identify new CHR constraints). The propagation history is used to avoid trivial non-termination that could be introduced by repeated application of the same instance of a propagation rule. The transition rules of ω_t are shown in Table 1.

Given a program P, the transition relation $\overset{\omega_t}{\rightarrow}_P\subseteq Conf\times Conf$ is the least relation satisfying the rules in Table 1. The *Solve* transition rule allows to update the constraint store by taking into account a built-in constraint contained in the goal. The *Introduce* transition rule is used to move a user-defined constraint from the goal to the CHR constraint store, where it can be handled by applying CHR rules. The *Apply* transition rule allows to rewrite user-defined constraints (which are in the CHR constraint store) by using rules from the program. As usual, in order to avoid variable name clashes, this transition rule assumes that all variables appearing in a program clause are fresh ones. The *Apply* transition rule is applicable when the current store (B) is strong enough to entail the guard of the rule (D), once the parameter passing has been performed. Note also that, as previously mentioned, the condition $\mathtt{id}(H_1) +\!\!+\ \mathtt{id}(H_2) +\!\!+\ [r]\notin T$ which avoids repeated application of the same instance of a propagation rule and therefore trivial non-termination.

An *initial configuration* has the form $\langle G,\emptyset,true,\emptyset\rangle_1$ while a *final configuration* has either the form $\langle G,S,false,T\rangle_k$, when it is *failed*, or the form $\langle\emptyset,S,B,T\rangle_k$, when it is successfully terminated because there are no applicable rules.

Given a goal G, the operational semantics that we consider observes the non-failed final stores of terminating computations. This notion of observable is the most used in the CHR literature and is captured by the following.

Definition 3 *(Qualified answers [7]).* Let P be a program and let G be a goal. The set $\mathcal{QA}_P(G)$ of qualified answers for the query G in the program P is defined as:

$$\mathcal{QA}_P(G)=\Big\{\exists_{-Fv(G)}(K\wedge d)\ \Big|\ \mathcal{CT}\not\models d\leftrightarrow false,\ \langle G,\emptyset,true,\emptyset\rangle_1 \overset{\omega_t}{\rightarrow}_P{}^{*}\ \langle\emptyset,K,d,T\rangle_n \overset{\omega_t}{\not\rightarrow}_P\Big\}$$

We also consider the following different notion of answer, obtained by computations terminating with a user-defined constraint which is empty. We call these observables *data sufficient answers* slightly deviating from the terminology of [7] (a goal which has a data sufficient answer is called a data sufficient goal in [7]).

Definition 4 *(Data sufficient answers).* Let P be a program and let G be a goal. The set $\mathcal{SA}_P(G)$ of data sufficient answers for the query G in the program P is defined as:

$$\mathcal{SA}_P(G)=\Big\{\exists_{-Fv(G)}(d)\ \Big|\ \mathcal{CT}\not\models d\leftrightarrow false,\ \langle G,\emptyset,true,\emptyset\rangle_1 \overset{\omega_t}{\rightarrow}_P{}^{*}\ \langle\emptyset,\emptyset,d,T\rangle_n\Big\}$$

Both previous notions of observables characterize an input/output behavior, since the input constraint is implicitly considered in the goal. Clearly in general $\mathcal{SA}_P(G)\subseteq\mathcal{QA}_P(G)$ holds, since data sufficient answers can be obtained by setting $K=\emptyset$ in Definition 3.

2.4. CHR with priorities

De Koninck et al. [3] extended CHR with user-defined priorities. This new language, denoted by *CHR*^{rp}, provides an high-level alternative for controlling program execution that is more appropriate for the needs of CHR programmers than other low-level approaches.

The syntax of CHR with priorities is compatible with the syntax of CHR. A simpagation rule has now the form

$$p :: r\ @H^k\backslash H^h \Longleftrightarrow D \mid B$$

where r, H^k, H^h, D, B are defined as in the CHR simpagation rule in Section 2.2, while p is an arithmetic expression, with $Fv(p)\subseteq(Fv(H^k)\cup Fv(H^h))$, which expresses the priority of rule r. If $Fv(p)=\emptyset$ then p is a static priority, otherwise it is called dynamic.

The formal semantics of *CHR*^{rp}, defined by [3], is an adaptation of the traditional semantics to deal with rule priorities. Formally this semantics, denoted by ω_p, is a state transition system $T=(Conf,\overset{\omega_p}{\rightarrow}_P)$ where P is a *CHR*^{rp} program while

$$
\begin{gathered}
1 :: \mathtt{source}(V) \Longrightarrow \mathtt{dist}(V, 0) \\
1 :: \mathtt{dist}(V, D_1) \backslash \mathtt{dist}(V, D_2) \Longleftrightarrow D_1 \leqslant D_2 | true \\
D + 2 :: \mathtt{dist}(V, D), \mathtt{edge}(V, C, U) \Longrightarrow \mathtt{dist}(U, D + C)
\end{gathered}
$$

Fig. 2. A program for computing the shortest path in *CHR*^{rp}.

Table 2
Transition rules of ω_p.

Solve	$\langle (c, G), S, C, T\rangle_n \overset{\omega_p}{\rightarrow}_P \langle G, S, c \wedge C, T\rangle_n$ where c is a built-in constraint
Introduce	$\langle (k, G), S, C, T\rangle_n \overset{\omega_p}{\rightarrow}_P \langle G, \{k\#n\} \cup S, C, T\rangle_{n+1}$ where k is a CHR constraint
Apply	$\langle \emptyset, H_1 \cup H_2 \cup S, C, T\rangle_n \overset{\omega_p}{\rightarrow}_P \langle B, H_1 \cup S, \theta \wedge D \wedge C, T \cup \{t\}\rangle_n$ where P contains a (renamed apart) rule $p :: r\ @H_1' \backslash H_2' \Longleftrightarrow D \mid B$ and there exists a matching substitution θ s.t. $\mathtt{chr}(H_1) = \theta H_1'$, $\mathtt{chr}(H_2) = \theta H_2'$, $\mathcal{CT} \models C \rightarrow \exists_{-Fv(C)}(\theta \wedge D)$, θp is a ground arithmetic expression and $t = \mathtt{id}(H_1) +\!\!+\ \mathtt{id}(H_2) +\!\!+\ [r] \notin T$. Furthermore no rule of priority p' and substitution θ' exists with $\theta' p' < \theta p$ for which the above conditions hold

configurations in *Conf*, as well as the initial and final configurations, are the same as those introduced for the traditional semantics in Section 2.3. The transition relation $\overset{\omega_p}{\rightarrow}_P \subseteq Conf \times Conf$ is the least relation satisfying the rules in Table 2. The *Solve* and *Introduce* transition rules are equal to those defined for the traditional semantics. The *Apply* transition rule instead is modified in order to take into account priorities. In fact, a further condition is added which requires that a rule can be fired only if no other rule that can be applied has a smaller value for the priority annotation (as usual in many systems, smaller values correspond to higher priority; For simplicity in the following we use the terminology "higher" or "lower" priority rather than considering the values).

An example of a *CHR*^{rp} program (from [3]) is shown in Fig. 2. This program can be used to compute the length of the shortest path between a source node and all the other nodes in the graph. We assume that the source node n is defined by using the constraint *source*(n) and that the graph is represented by using the constraints *edge*(V, C, U) for every edge of length C between two nodes V and U. When the program terminates we obtain a constraint *dist*(U, C) iff the length of the shortest path between the source node and U is C.

The qualified and data sufficient answers for *CHR*^{rp} can be defined analogously to those of the standard language.

2.5. Language encoding

In this work we consider the following languages and semantics:

- *CHR*$^{\omega_t}$: this is standard CHR, where the theoretical semantics is used;
- *CHR*^{rp}: this is CHR with priorities, where both dynamic and static priorities can be used, the semantics is that one defined in the previous section (ω_p);
- *static CHR*^{rp}: this is CHR with static priorities only, with the ω_p semantics;
- *static CHR*$^{rp}_2$: this is CHR with static priorities only, with the ω_p semantics, where we allow at most two constraints in the head of a rule.

Since all these languages are Turing powerful [14] in principle one can always encode a language into another one. The question is how difficult and how natural such an encoding is. As mentioned in the introduction, depending on the answer to this question one can discriminate different languages. Indeed, several approaches which compare the expressive power of concurrent languages impose the condition that the translation is compositional w.r.t. some operator of the language, because compositionality is considered a natural property of the translation. Moreover, usually one wants that some observable properties of the computations are preserved by the translation, which is also a natural requirement.

In the following we then make similar assumptions on our encoding functions for CHR languages. We formally define a *program encoding* as any function $\mathcal{PROG} : \mathcal{P}_{\mathcal{L}} \rightarrow \mathcal{P}_{\mathcal{L}'}$ which translates an $\mathcal{L}$ program into a (finite) $\mathcal{L}'$ program ($\mathcal{P}_{\mathcal{L}}$ and $\mathcal{P}_{\mathcal{L}'}$ denote the set of $\mathcal{L}$ and $\mathcal{L}'$ programs, respectively). To simplify the treatment we assume that both the source and the target language use the same built-in constraints semantically described by a theory $\mathcal{CT}$. Next we have to define how the initial goal and the observables should be translated by the encoding and the decoding functions, respectively. We require that these translations are compositional w.r.t. the conjunction of atoms. This assumption essentially means that the encoding and the decoding respect the structure of the original goal and of the observables. Moreover, since the source and the translated programs use the same constraint theory, it is natural to assume also that these two functions do not modify or add built-in constraints (in other words, we do not allow to simulate the behavior and the effects of the constraint theory).

We do not impose any restriction on the program translation, hence we have the following definition.

Definition 5 *(Acceptable encoding).* Suppose that $\mathcal{C}$ is the class of all the possible multisets of constraints. An *acceptable encoding* (of $\mathcal{L}$ into $\mathcal{L}'$) is a tern of mappings $(\mathcal{PROG}, \mathcal{INP}, \mathcal{OUT})$, where $\mathcal{PROG} : \mathcal{P}_{\mathcal{L}} \to \mathcal{P}_{\mathcal{L}'}$ is the program encoding, $\mathcal{INP} : \mathcal{C} \to \mathcal{C}$ is the goal encoding, and $\mathcal{OUT} : \mathcal{C} \to \mathcal{C}$ is the output decoding, which satisfy the following conditions:

1. The goal encoding function is compositional, that is, for any goal $(A, B) \in \mathcal{C}$, $\mathcal{INP}(A, B) = \mathcal{INP}(A), \mathcal{INP}(B)$ holds. We also assume that the built-ins present in the goal are left unchanged and no new built-ins can be added;
2. The output decoding function is compositional, that is, for any qualified answer $(A, B) \in \mathcal{C}$, $\mathcal{OUT}(A, B) = \mathcal{OUT}(A), \mathcal{OUT}(B)$ holds. We also assume that the built-ins present in the answer are left unchanged and no new built-ins can be added;
3. Qualified answers are preserved for the class $\mathcal{C}$, that is, for all $P \in \mathcal{P}_{\mathcal{L}}$ and $G \in \mathcal{C}$, $\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\mathcal{PROG}(P)}(\mathcal{INP}(G)))$ holds.

Moreover we define an *acceptable encoding for data sufficient answers* of $\mathcal{L}$ into $\mathcal{L}'$ exactly as an acceptable encoding, with the exception that the third condition above is replaced by the following:

3′. Data sufficient answers are preserved for the class $\mathcal{C}$, that is, for all $P \in \mathcal{P}_{\mathcal{L}}$ and $G \in \mathcal{C}$, $\mathcal{SA}_P(G)$ is equal to the data sufficient answers in $\mathcal{OUT}(\mathcal{QA}_{\mathcal{PROG}(P)}(\mathcal{INP}(G)))$.[2]

Further weakening these conditions and requiring, for instance, that the translation of A, B is some form of composition (rather than the conjunction) of the translation of A and B does not seem reasonable, as conjunction is the only form for goal composition available in CHR.

Note that, according to the previous definition, if there exists an acceptable encoding then there exists also an acceptable encoding for data sufficient answers. This is an immediate consequence of the fact that data sufficient answers are a subset of qualified answers.

In the following, given a program P, we denote by $Pred(P)$ and $Head(P)$ the set of all the predicate symbols p s.t. p occurs in P and in the head of a rule in P, respectively.

3. Positive results

In this section we present some (acceptable) encodings for the four languages described at the beginning of Section 2.5. We first present some immediate results which derive directly from the language definitions. Then we describe two of the main results of this paper, namely that there exists acceptable encodings from *static CHR*^{rp} to *static CHR*$^{rp}_2$ and from *CHR*^{rp} to *static CHR*^{rp}. The combination of these results shows that *static CHR*$^{rp}_2$ is as powerful as the full *CHR*^{rp}, that is, a program with dynamic priorities can be (acceptably) encoded into one with static priorities and this, in its turn, can be encoded into a program which does not use more than two constraints in the head of rules.

We first observe that *static CHR*$^{rp}_2$ is a sublanguage of *static CHR*^{rp} that, in its turn, is a sublanguage of *CHR*^{rp}. Moreover, when a language $\mathcal{L}$ is a sublanguage of $\mathcal{L}'$ then a tern of identity functions provides an acceptable encoding between the two languages. Therefore we have the following.

Fact 1. There exist acceptable encodings from *static CHR*$^{rp}_2$ to *static CHR*^{rp}, and from *static CHR*^{rp} to *CHR*^{rp}.

As far as *CHR*$^{\omega_t}$ is concerned, at a first glance it could be considered as a sublanguage of *static CHR*^{rp} where all the rules have equal priority. However this is not completely true since in the ω_p semantics, for the application of an Apply transition, the goal multiset of the configuration must be empty while in the ω_t semantics it is possible to fire a rule even though some constraints have not being introduced into the CHR store by a Solve or an Introduce transition. However it is easy to see that from the monotonicity of ω_t it follows that for every computation reaching a non-failed final configuration there is one computation reaching the same final configuration where the Solve and Introduce transitions are performed as soon as they can be executed. Hence every final configuration reached by a *CHR*$^{\omega_t}$ program P can be reached by the *static CHR*^{rp} program having the same rules as P with a fixed and constant priority. Therefore we have the following.

Fact 2. There exists an acceptable encoding from *CHR*$^{\omega_t}$ to *static CHR*^{rp}.

As previously mentioned, the existence of an acceptable encoding implies the existence of an acceptable encoding for data sufficient answers. Hence we have the following immediate corollary.

[2] Note that in 3. and in 3′. the function $\mathcal{OUT}$ is extended in the obvious way to sets of qualified answers.

Corollary 1. *There exist acceptable encodings for data sufficient answers from CHR$^{\omega_t}$ to static CHR^{rp}, from static CHR$_2^{rp}$ to static CHR^{rp}, and from static CHR^{rp} to CHR^{rp}.*

3.1. *Encoding static CHR^{rp} into static CHR$_2^{rp}$*

In this section we provide an acceptable encoding from *static CHR^{rp}* to *static CHR$_2^{rp}$*. We assume that P is a *static CHR^{rp}* program composed by m rules and that the i-th rule (with $i \in \{1,\dots,m\}$) has the form:

$$p_i :: rule_i \ @ \ h_{(i,1)}(\bar{t}_1),\dots,h_{(i,l_i)}(\bar{t}_{l_i}) \backslash h_{(i,l_i+1)}(\bar{t}_{l_{i+1}}),\dots,h_{(i,r_i)}(\bar{t}_{r_i}) \Leftrightarrow G_i|C_i$$

Moreover we denote by p_{max} the lowest priority (i.e. the biggest p_i).

First, we require that the goal encoding (the second component of our acceptable encoding) is a non-surjective function. The reason for this requirement is that the program encoding (first component of the triple) needs to use, in the translated program, some fresh constraints which do not appear in the initial (translated) goal. A simple goal encoding that satisfies this requirement is the one that does not change built-in constraints and adds a letter, say “a”, at the beginning of the other constraints, as shown below

$$\mathcal{INP}\big(b(\bar{t})\big) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ ab(\bar{t}) & \text{otherwise} \end{cases}$$

In the rest of this section, by a slight abuse of notation, we use the notation $\mathcal{INP}$ also to indicate a function from predicate symbols to predicate symbols.

The main ingredient of an acceptable encoding is the program encoding. In the following we define an encoding that produces a program that simulates the execution of a *static CHR^{rp}* by using only rules having static priorities and at most two constraints in the head.

Intuitively, the constraint identifier introduced by an Introduce transition rule in the original program is simulated by adding a unique term as an argument to a new kind of constraint.

The simulation process can be divided in the following three phases:

1. *Initialization*. In the initialization phase, for each $k \in \mathcal{INP}(Head(P))$ we replace a constraint $k(\bar{t})$ by $new_k(n,\bar{t})$ where n is a fresh constant. This allows the encoded program to simulate the Introduce transition rules.
2. *Main*. In the main phase the encoded program determines what rules of the original program can fire. If there is one rule that can fire its firing is simulated and then all the constraints that are used to simulate the firing of the rule are deleted. This phase is repeated until no rule of the original program can fire.
3. *Termination*. The termination phase starts at the end of the main phase. In this case all the constraints produced during the computation for the simulation purposes are deleted.

In order to define the program encoding we introduce the following constraints:

- $id(t)$; used to simulate an Introduce transition rule step; t is a term that will be used as a constraint identifier.
- end; used to delete the constraint added in the process of simulating the firing of the rules.
- $rC[N]_i(\bar{t})$ with $N \in \{1,\dots,r_i\}$ where r_i is the number of constraints in the head of the i-th rule; used to check if the rule $rule_i$ can fire.
- $rA_i(\bar{t})$ is a constraint which is added to the store when the i-th rule is fired; the $\bar{t}$ are the identifiers of the constraints which are consumed by the application of the i-th rule and therefore should be removed from the store.
- $new_k(V,\bar{u})$ where V is a term, $\bar{u}$ is a sequence of terms and k is a predicate symbol in $\mathcal{INP}(Pred(P))$; this new constraint will be used to add to a constraint $k(\bar{u})$ a new identifier V.

Note that since no constraint in this list starts with an “a”, the previous assumption on the goal encoding function $\mathcal{INP}$ implies that these constraints cannot be in any goal produced by $\mathcal{INP}$.

In the following, in order to simplify the notation, when we are not interested in the arguments of a predicate we simply use an underscore to indicate them (thus writing, for example, $p(\_), q(\_)$).

Formally the program encoding, denoted by α, is a function that given a *static CHR^{rp}* program P, returns the program constructed as follows:

$$\begin{gathered} \text{for every predicate name } k \in \mathcal{INP}(Head(P)) \\ 1 :: rule_{(1,k)} \ @ \ id(V), k(\bar{X}) \Leftrightarrow id(V+1), new_k(V,\bar{X}) \\ 2 :: rule_{(2,k)} \ @ \ k(\bar{X}) \Leftrightarrow id(2), new_k(1,\bar{X}) \end{gathered}$$

$$\begin{gathered} \text{for every } i \in \{1,\dots,m\},\ N \in [2, r_i - 1] \\ 3 :: rule_{(3,i,N)} \ @ \ end \backslash rC[N]_i(\_) \Leftrightarrow true \end{gathered}$$

$$\begin{gathered} \text{for every predicate name } k \in \mathcal{INP}(Head(P)),\ i \in \{1,\dots,m\} \\ 3 :: rule_{(4,i,k)} \ @ \ rA_i(\bar{V}) \backslash new_k(V',\bar{X}) \Leftrightarrow V' \in \bar{V} | true \end{gathered}$$

$$\text{for every } i, j \in \{1,\dots,m\},\ N \in [2, r_i - 1]$$
$$3 :: rule_{(5,j,i,N)}\ @\ rA_j(\bar{V}) \backslash rC[N]_i(\bar{V}', \bar{X}) \Leftrightarrow \bar{V} \cap \bar{V}' \neq \emptyset | true$$

$$\text{for every } i \in \{1,\dots,m\}$$
$$4 :: rule_{(6,i)}\ @\ rA_i(\_) \Leftrightarrow true$$

$$\text{for every } i \in \{1,\dots,m\} CHECK\_RULE(i)$$

$$\text{for every } i \in \{1,\dots,m\}$$
$$6 + p_i :: rule_{(7,i)}\ @\ rC[r_i]_i(V_1,\dots,V_{r_i}, \bar{t}_1,\dots,\bar{t}_{r_i}), id(V) \Leftrightarrow$$
$$G_i | Update(\mathcal{INP}(C_i), V), rA_i(V_{l_i+1},\dots,V_{r_i})$$

$$7 + p_{max} :: rule_8\ @\ id(\_) \Leftrightarrow end$$
$$7 + p_{max} :: rule_9\ @\ end \Leftrightarrow true$$

where $CHECK\_RULE(i)$ are the following rules

$$\text{for every } N \in [2, r_i]$$
$$5 :: rule'_{(i,N)}\ @\ rC[N-1]_i(\bar{V}_1, \bar{X}_1), new_{\mathcal{INP}(h_{(i,N)})}(V_2, \bar{X}_2) \Rightarrow$$
$$V_2 \notin \bar{V}_1 | rC[N]_i(\bar{V}_1, V_2, \bar{X}_1, \bar{X}_2)$$

where by convention, $rC[1]_i(V, \bar{X}) = new_{\mathcal{INP}(h_{(i,1)})}(V, \bar{X})$ and $Update(C, V)$ is defined as follows

$$Update(k(\bar{t}), V) = new_k(V, \bar{t})$$
$$\text{if } k(\bar{t}) \text{ is a CHR constraint}$$

$$Update(c(\bar{t}), V) = c(\bar{t})$$
$$\text{if } c(\bar{t}) \text{ is a built-in constraint}$$

$$Update([\ ], V) = id(V)$$

$$Update([d(\bar{X}) \mid Ds], V) =$$
$$Update(d(\bar{X}), V), Update(Ds, V+1).$$

Example 1. As an example for the application of the program encoding α let us consider the simple program P composed by the following rule:

$$1 :: h_1(X), h_2(Y) \backslash h'(Z) \Leftrightarrow X = Y | h$$

$\alpha(P)$ is the following program:

$$1 :: rule_{(1,ah_1)}\ @\ id(V), ah_1(X) \Leftrightarrow id(V+1), new_{ah_1}(V, X)$$
$$1 :: rule_{(1,ah_2)}\ @\ id(V), ah_2(X) \Leftrightarrow id(V+1), new_{ah_2}(V, X)$$
$$1 :: rule_{(1,ah')}\ @\ id(V), ah'(X) \Leftrightarrow id(V+1), new_{ah'}(V, X)$$

$$2 :: rule_{(2,ah_1)}\ @\ ah_1(X) \Leftrightarrow id(2), new_{ah_1}(1, X)$$
$$2 :: rule_{(2,ah_2)}\ @\ ah_2(X) \Leftrightarrow id(2), new_{ah_2}(1, X)$$
$$2 :: rule_{(2,ah')}\ @\ ah'(X) \Leftrightarrow id(2), new_{ah'}(1, X)$$

$$3 :: rule_{(3,1,2)}\ @\ end \backslash rC2_1(V_1, V_2, X_1, X_2) \Leftrightarrow true$$

$$3 :: rule_{(4,1,ah_1)}\ @\ rA_1(V) \backslash new_{ah_1}(V', X) \Leftrightarrow V' = V | true$$
$$3 :: rule_{(4,1,ah_2)}\ @\ rA_1(V) \backslash new_{ah_2}(V', X) \Leftrightarrow V' = V | true$$
$$3 :: rule_{(4,1,ah')}\ @\ rA_1(V) \backslash new_{ah'}(V', X) \Leftrightarrow V' = V | true$$

$$3 :: rule_{(5,1,1,1)}\ @\ rA_1(V) \backslash rC2_1(V_1, V_2, X_1, X_2) \Leftrightarrow$$
$$V \notin \{V_1, V_2\} | true$$
$$3 :: rule_{(5,1,1,2)}\ @\ rA_1(V) \backslash rC3_1(V_1, V_2, V_3, X_1, X_2, X_3) \Leftrightarrow$$
$$V \notin \{V_1, V_2, V_3\} | true$$

$$4 :: rule_{(6,1)}\ @\ rA_1(V) \Leftrightarrow true$$

$$5 :: rule'_{(1,2)}\ @\ new_{ah_1}(V_1, X_1), new_{ah_2}(V_2, X_2) \Rightarrow$$
$$V_2 \neq V_1 | rC2_1(V_1, V_2, X_1, X_2)$$
$$5 :: rule'_{(1,3)}\ @\ rC2_1(V_1, V_2, X_1, X_2), new_{ah'}(V_3, X_3) \Rightarrow$$
$$V_3 \notin \{V_1, V_2\} | rC3_1(V_1, V_2, V_3, X_1, X_2, X_3)$$

$$7 :: rule_{(7,1)} \ @ \ rC3_1(V_1, V_2, V_3, X, Y, Z), id(V) \Leftrightarrow$$
$$X = Y | new_{ah}(V), id(V+1), rA_1(V_3)$$

$$8 :: rule_8 \ @ \ id(V) \Leftrightarrow end$$
$$8 :: rule_9 \ @ \ end \Leftrightarrow true$$

In the following, in order to better clarify the execution order of the rules, we describe how they are used in the three phases of the encoded program.

1. *Initialization*. The first rule to be fired is a rule $rule_{(2,k)}$ that triggers the firing of rules $rule_{(1,k)}$. This allows the replacing of every constraint k in $\mathcal{INP}(Head(P))$ by the constraint $new_k(n, \bar{t})$. The predicate symbol id is used to memorize the highest identifier used.
2. *Main*. The main phase can be divided into three sub-phases. The first sub-phase is the *evaluation* that starts when the init phase terminates (at this point all the constraints $k(\bar{t})$, with $k \in \mathcal{INP}(Head(P))$ have been converted into $new_k(l, \bar{t})$). Rules $rule'_{(i,N)}$ determine what rules belonging to the original program can fire. The second sub-phase is the *activation*. During this sub-phase if $rule_i$ can be fired in the original program P then $rule_{(7,i)}$ can be fired in the program $\alpha(P)$. If the original program has not reached the final state then one of the rules $rule_{(7,i)}$ fires, starting the *deletion* sub-phase. In this last sub-phase rules $rule_{(4,i,k)}$, $rule_{(5,j,i,N)}$ and $rule_{(6,i)}$ delete all the constraints that are used to simulate the constraints deleted by the application of the i-th rule in the original program P.
3. *Termination*. The termination phase is triggered by rule $rule_8$ that is used to detect when no rule $rule_{(7,i)}$ can fire (this happens iff the original program has reached a final state). Rules $rule_{(3,i,N)}$ and $rule_9$ delete all the constraints produced during the computation for the simulation purpose, that is id, $rC[N]_i$ and end.

It is worth noting that the operators $\in$, $\notin$ and $\cap$ written in the guards can be replaced by equalities and inequalities. In rules $rule'_{(i,N)}$, for instance, the guards $V' \notin \bar{V}$, where $\bar{V} = V_1, \ldots, V_n$ can be replaced by $V' \neq V_1, \ldots, V' \neq V_n$. Rules

$$3 :: rule_{(4,i,k)} \ @ \ rA_i(\bar{V}) \backslash new_k(V', \bar{X}) \Leftrightarrow V' \in \bar{V} | true$$

where $\bar{V} = V_{l_{i+1}}, \ldots, V_{r_i}$ can be rewritten by the set of rules

$$\{3 :: rule_{(4,i,k,o)} \ @ \ rA_i(\bar{V}) \backslash new_k(V', \bar{X}) \Leftrightarrow V' = V_o | true \mid o \in [l_{i+1}, r_i]\}$$

and finally rules

$$3 :: rule_{(5,j,i,N)} \ @ \ rA_j(\bar{V}) \backslash rC[N]_i(\bar{V}', \bar{X}) \Leftrightarrow \bar{V} \cap \bar{V}' \neq \emptyset | true$$

where $\bar{V} = V_{l_{i+1}}, \ldots, V_{r_i}$ and $\bar{V}' = V'_1, \ldots, V'_N$ can be rewritten by the set of rules

$$\{3 :: rule_{(5,j,i,N,o,p)} \ @ \ rA_j(\bar{V}) \backslash rC[N]_i(\bar{V}', \bar{X}) \Leftrightarrow V_o = V'_p | true \mid o \in [l_{i+1}, r_i] \text{ and } p \in [1, N]\}$$

Even though it is common to have a constraint theory $\mathcal{CT}$ supporting the equality and inequality as built-ins, this assumption is not technically needed and the α encoding can be obtained also without the use of equalities and inequalities (for more details on this point please see Appendix A).

To conclude the definition of the acceptable encoding we need the last ingredient: the output decoding function. If we run the goal $\mathcal{INP}(G)$ in the program $\alpha(P)$ we obtain the same qualified answers obtained by running G in the program P, with the only difference that if in the qualified answer of P there is a CHR constraint $k(\bar{t})$ then in the corresponding qualified answer of the encoded program $\alpha(P)$ there will be either a constraint $new_{ak}(V, \bar{t})$ (if $k \in Head(P)$ or $k(\bar{t})$ is introduced by an Apply transition rule step) or a constraint $ak(\bar{t})$ (if $k \notin Head(P)$ and $k(\bar{t})$ is in the initial goal G).

Therefore the decoding function that we need is:

$$\mathcal{OUT}(b(\bar{t})) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ k(\bar{t}') & \text{if } b(\bar{t}) = new_{ak}(V, \bar{t}') \\ k(\bar{t}) & \text{if } b(\bar{t}) = ak(\bar{t}) \end{cases}$$

The following theorem, whose proof is in Appendix B, shows that the triple $(\alpha, \mathcal{INP}, \mathcal{OUT})$ that we have defined indeed provides the desired encoding.

Theorem 1. *The triple $(\alpha, \mathcal{INP}, \mathcal{OUT})$ provides an acceptable encoding from static CHR^{rp} into static CHR$^{rp}_2$.*

Note that, as mentioned after Definition 5, if there exists an acceptable encoding then there exists also an acceptable encoding for data sufficient answers. Hence the previous result implies that there is an acceptable encoding for data sufficient answers from *static CHR^{rp}* into *static CHR$^{rp}_2$*.

Moreover, if data sufficient answers are considered it is possible to strengthen the previous theorem by requiring that the goal encoding and the output decoding functions are the identity functions. This does not hold if we consider the program encoding α presented in the previous session. Intuitively the reason is that when the goal encoding function is the identity function, such constraints as $id, end, rC[N]_i$ could be in the initial goal of the encoded program. However, when we are focusing on data sufficient answers, we can overcome this problem and use the same program encoding as a base for a new program encoding for data sufficient answers. For the interested reader the definition of such an encoding is presented in Appendix C.

3.2. Encoding CHR^{rp} into static CHR^{rp}

In this section we prove that the *CHR^{rp}* language, which allows dynamic priorities, is not more expressive than *static CHR^{rp}*, which allows static priorities only. This result is obtained by providing an (acceptable) encoding of *CHR^{rp}* into *static CHR^{rp}*.

As usual, we assume that P is a *CHR^{rp}* program composed of m rules and we also assume that the i-th rule (with $i \in \{1, \dots, m\}$) has the form:

$$p_i :: rule_i \ @ \ H_i \backslash H_i' \Leftrightarrow G_i | B_i$$

Moreover, given a multiset of CHR constraints $\bar{H} = h_1(\bar{t}_1), \dots, h_n(\bar{t}_n)$ and a sequence of (distinct) variables $\bar{V} = V_1, \dots, V_n$, we denote by $new'(\bar{H}, \bar{V})$ the multiset of atoms $new_{h_1}(V_1, \bar{t}_1), \dots, new_{h_n}(V_n, \bar{t}_n)$.

As for the goal encoding and the output decoding functions we use here the same functions $\mathcal{INP}$ and $\mathcal{OUT}$ defined in Section 3.1.[3]

In the following, we define the program encoding $\mathcal{T}(P)$ from *CHR^{rp}* into *static CHR^{rp}* that produces a program that simulates the execution of a *CHR^{rp}* program with only rules having static priorities. The simulation process can be divided in the following three phases:

1. *Initialization*. In the initialization phase, for each (user-defined) predicate symbol $ak \in \mathcal{INP}(Head(P))$ we replace a constraint $ak(\bar{t})$ by $new_{ak}(V, \bar{t})$ where V is a new variable. This allows the encoded program to simulate the Introduce transition rules.
2. *Main*. The main phase consists of the rules EVALUATE_PRIORITIES(i) and ACTIVATE_RULE(i), for $i \in \{1, \dots, m\}$. The first set of rules determines what rules of the original program can fire, while the rules of the second set are used to simulate the firing of the original rules that can fire.
3. *Termination*. The termination phase starts at the end of the main phase. In this case all the constraints produced during the computation for the simulation purposes are deleted.

Formally, the program encoding, denoted by $\mathcal{T}(P)$, is a function that given a *CHR^{rp}* program P, returns the program constructed as follows:

$$\begin{gathered}
\text{for every predicate name } ak \in \mathcal{INP}(Head(P)) \\
1 :: rule_{(1,k)} \ @ \ start \backslash id(V), ak(\bar{X}) \Leftrightarrow id(V+1), new_{ak}(V, \bar{X}) \\
2 :: rule_{(2,k)} \ @ \ ak(\bar{X}) \Rightarrow start, id(0)
\end{gathered}$$

$$2 :: rule_3 \ @ \ start \Leftrightarrow highest\_priority(inf)$$

$$\begin{gathered}
\text{for every } i \in \{1, \dots, m\} \\
3 :: rule_{(4,i)} \ @ \ end \backslash instance_i(\_) \Leftrightarrow true
\end{gathered}$$

$$4 :: rule_5 \ @ \ end \Leftrightarrow true$$

$$\text{for every } i \in \{1, \dots, m\} \quad \texttt{EVALUATE\_PRIORITIES}(i)$$

$$7 :: rule_9 \ @ \ highest\_priority(inf), id(V) \Leftrightarrow end$$

$$\text{for every } i \in \{1, \dots, m\} \quad \texttt{ACTIVATE\_RULE}(i)$$

If $rule_i$ is not a propagation rule then EVALUATE_PRIORITIES(i) are the following rules

$$\begin{aligned}
6 :: rule_{(7,i)} \ @ \ & new'(\mathcal{INP}(H_i), \bar{Z}), new'(\mathcal{INP}(H'_i), \bar{U}) \backslash highest\_priority(inf) \\
& \Leftrightarrow G_i | highest\_priority(p_i)
\end{aligned}$$

$$\begin{aligned}
6 :: rule_{(8,i)} \ @ \ & new'(\mathcal{INP}(H_i), \bar{Z}), new'(\mathcal{INP}(H'_i), \bar{U}) \backslash highest\_priority(P) \\
& \Leftrightarrow G_i, p_i < P | highest\_priority(p_i)
\end{aligned}$$

if $rule_i$ is a propagation rule then EVALUATE_PRIORITIES(i) are the following rules

[3] In the rest of this section, by a slight abuse of notation, we use the function $\mathcal{INP}$ also as a function from predicate symbols to predicate symbols.

$$5 :: rule_{(6,i)}\ @new'(\mathcal{INP}(H_i), \bar{Z}) \Rightarrow G_i | instance_i(\bar{Z})$$

$$6 :: rule_{(7,i)}\ @\ instance_i(\bar{Z}), new'(\mathcal{INP}(H_i), \bar{Z}) \backslash highest\_priority(inf) \Leftrightarrow G_i | highest\_priority(p_i)$$

$$6 :: rule_{(8,i)}\ @\ instance_i(\bar{Z}), new'(\mathcal{INP}(H_i), \bar{Z}) \backslash highest\_priority(P) \Leftrightarrow G_i, p_i < P | highest\_priority(p_i)$$

if $rule_i$ is a propagation rule then ACTIVATE_RULE(i) is the following rule

$$8 :: rule_{(10,i)}\ @\ new'(\mathcal{INP}(H_i), \bar{Z}) \backslash instance_i(\bar{Z}), highest\_priority(P), id(V) \Leftrightarrow G_i, p_i = P | Update(\mathcal{INP}(B_i), V), highest\_priority(inf)$$

if $rule_i$ is not a propagation rule then ACTIVATE_RULE(i) is the following rule

$$8 :: rule_{(10,i)}\ @\ new'(\mathcal{INP}(H_i), \bar{Z}) \backslash new'(\mathcal{INP}(H'_i), \bar{U}), highest\_priority(P), id(V) \Leftrightarrow G_i, p_i = P | Update(\mathcal{INP}(B_i), V), highest\_priority(inf)$$

In the above encoding we assume that the constraint theory $\mathcal{CT}$ allows to use equalities and inequalities (so we can evaluate whether $p_i = h$ and $p_i > h$ where $h \in \mathbb{Z}$ and p_i is an arithmetic expression). We also assume *inf* is a conventional constant which is bigger than all p_i (i.e. it represents the lowest priority). The *Update* function is exactly the one defined in Section 3.1.

Example 2. Let us consider as P the shortest path program depicted in Fig. 2. The corresponding $\mathcal{T}(P)$ is the following program:

$$1 :: rule_{(1,asource)}\ @\ start \backslash id(V), asource(\bar{X}) \Leftrightarrow id(V+1), new_{asource}(V, \bar{X})$$
$$1 :: rule_{(1,adist)}\ @\ start \backslash id(V), adist(\bar{X}) \Leftrightarrow id(V+1), new_{adist}(V, \bar{X})$$
$$1 :: rule_{(1,aedge)}\ @\ start \backslash id(V), aedge(\bar{X}) \Leftrightarrow id(V+1), new_{aedge}(V, \bar{X})$$

$$2 :: rule_{(2,asource)}\ @\ asource(\bar{X}) \Rightarrow start, id(0)$$
$$2 :: rule_{(2,adist)}\ @\ adist(\bar{X}) \Rightarrow start, id(0)$$
$$2 :: rule_{(2,aedge)}\ @\ aedge(\bar{X}) \Rightarrow start, id(0)$$

$$2 :: rule_{(3)}\ @\ start \Leftrightarrow highest\_priority(inf)$$

$$3 :: rule_{(4,1)}\ @\ end \backslash instance_1(\bar{Z}) \Leftrightarrow true$$
$$3 :: rule_{(4,2)}\ @\ end \backslash instance_2(\bar{Z}) \Leftrightarrow true$$
$$3 :: rule_{(4,3)}\ @\ end \backslash instance_3(\bar{Z}) \Leftrightarrow true$$

$$4 :: rule_5\ @\ end \Leftrightarrow true$$

$$5 :: rule_{(6,1)}\ @\ new_{asource}(V, X) \Rightarrow instance_1(V)$$
$$6 :: rule_{(7,1)}\ @\ new_{asource}(V, X) \backslash highest\_priority(inf) \Leftrightarrow highest\_priority(1)$$
$$6 :: rule_{(8,1)}\ @\ new_{asource}(V, X) \backslash highest\_priority(P) \Leftrightarrow 1 < P | highest\_priority(1)$$

$$6 :: rule_{(7,2)}\ @\ new_{adist}(V_1, X_1, X_2), new_{adist}(V_2, Y_1, Y_2) \backslash highest\_priority(inf) \Leftrightarrow X_2 \leqslant Y_2 | highest\_priority(1)$$
$$6 :: rule_{(8,2)}\ @\ new_{adist}(V_1, X_1, X_2), new_{adist}(V_2, Y_1, Y_2) \backslash highest\_priority(P) \Leftrightarrow X_2 \leqslant Y_2, 1 < P | highest\_priority(1)$$

$$5 :: rule_{(6,3)}\ @\ new_{adist}(V_1, X_1, X_2), new_{aedge}(V_2, \bar{Y}) \Rightarrow instance_3(V_1, V_2)$$
$$6 :: rule_{(7,3)}\ @\ new_{adist}(V_1, X_1, X_2), new_{aedge}(V_2, \bar{Y}) \backslash highest\_priority(inf) \Leftrightarrow highest\_priority(X_2 + 2)$$
$$6 :: rule_{(8,3)}\ @\ new_{adist}(V_1, X_1, X_2), new_{aedge}(V_2, \bar{Y}) \backslash highest\_priority(P) \Leftrightarrow X_2 + 2 < P | highest\_priority(X_2 + 2)$$

$$7 :: rule_9\ @\ highest\_priority(inf), id(V) \Leftrightarrow end$$

$$8 :: rule_{(10,1)}\ @\ new_{asource}(V, X) \backslash instance_1(V), highest\_priority(P), id(V')$$

$$\Leftrightarrow 1 = P|new_{adist}(V', X, 0), id(V'+1), highest\_priority(inf)$$
$$8 :: rule_{(10,2)} \ @ \ new_{adist}(V_1, X, X_1)\backslash new_{adist}(V_2, X, X_2), highest\_priority(P), id(V')$$
$$\Leftrightarrow X_1 \leqslant X_2, 1 = P|id(V'), highest\_priority(inf)$$
$$8 :: rule_{(10,3)} \ @ \ new_{adist}(V_1, X, X_1), new_{aedge}(V_2, X, X_2, X_3)\backslash instance_3(V_1, V_2),$$
$$highest\_priority(P), id(V') \Leftrightarrow X_1 + 2 = P|new_{adist}(V', X_3, X_1 + X_2),$$
$$id(V'+1), highest\_priority(inf)$$

In the following, to better clarify the execution order of the rules, we describe how they are used in the three phases of the encoded program.

1. *Initialization*. In the init phase, for each (user-defined) predicate symbol $ak \in \mathcal{INP}(Head(P))$ we introduce a rule $rule_{(1,k)}$, which replaces $ak(\bar{t})$ by $new_{ak}(V, \bar{t})$ where V is a variable which will be used to simulate the identifier used in identified constraints. Moreover we use the *id* predicate symbol to memorize the highest identifier used. Rules $rule_{(2,k)}$ (one for each predicate symbol $ak \in \mathcal{INP}(Head(P))$, as before) are used to fire rules $rule_{(1,k)}$ and also to start the following phase (via $rule_3$). Note that rules $rule_{(1,k)}$ have maximal priority and therefore are tried before rules $rule_{(2,k)}$.
2. *Main*. The main phase is divided into two phases: the *evaluation* phase starts when the init phase adds the constraint $highest\_priority(inf)$. Rules $rule_{(6,i)}, \ldots, rule_{(8,i)}$ store in *highest_priority* the highest priority on all the rule instances that can be fired. After the end of the evaluation phase the *activation* starts. During this phase if a rule can be fired one of the rules $rule_{(10,i)}$ is fired. After the rule has been fired the constraint $highest\_priority(inf)$ is produced which starts a new evaluation phase.
3. *Termination*. The termination phase is triggered by rule $rule_9$. This rule fires when no instance from the original program can fire. During the termination phase all the constraints produced during the computation (namely *id*, $instance_i$, *highest_priority*, *end*) are deleted.

In the following we now provide some more details on the two crucial points in this translation: the evaluation and the activation phases.

- *Evaluation*. The rules in the set denoted by

  EVALUATE_PRIORITIES(i)

  are triggered by the insertion of $highest\_priority(inf)$ in the constraint store.
  In the case of a propagation rule $rule_i \in P$, the rules in

  EVALUATE_PRIORITIES(i)

  should consider the possibility that there is an instance of $rule_i$ that cannot be fired because it has been previously fired. When an instance of a propagation rule can fire, rule $rule_{(6,i)}$ adds a constraint $instance_i(\bar{v})$, where $\bar{v}$ are the identifiers of the CHR atoms which can be used to fire $rule_i$. The absence of the constraint $instance_i(\bar{v})$ in the constraint store means that either $rule_i$ cannot be fired by using the CHR atoms identified by $\bar{v}$ or has already fired for the CHR atoms identified by $\bar{v}$.
  The evaluation of the priority for a simpagation or a simplification rule is instead more simple because the propagation history does not affect the execution of these two types of rules.
  Rules $rule_{(7,i)}$ and $rule_{(8,i)}$ replace the constraint $highest\_priority(p)$ with the constraint $highest\_priority(p')$ if a rule of priority p' can be fired and $p > p'$.
- *Activation*. When the evaluation phase ends, if a rule can fire then one of the rules $rule_{(10,i)}$ is fired since $highest\_priority(inf)$ has been removed from the constraint store.
  The only difference between a propagation rule and a simpagation/simplification rule is that when a propagation rule is fired the corresponding constraint $instance_i(\bar{v})$ is deleted to avoid the execution of the same propagation rule in the future.
  It is worth noting that the non-determinism in the choice of the rule to be fired provided by the ω_p semantics is preserved, since all the priorities of ACTIVATE_RULE(i) are equal.

The following result shows that the qualified answers are preserved by our encoding. Its proof, in Appendices A, B, C and D, follows the lines of the reasoning informally explained above. The functions $\mathcal{INP}$ and $\mathcal{OUT}$ are those defined in Section 3.1, while $\mathcal{T}$ is defined before.

Theorem 2. *The triple $(\mathcal{T}, \mathcal{INP}, \mathcal{OUT})$ provides an acceptable encoding between CHR^{rp} and static CHR^{rp}.*

Analogously to the case of previous section, previous result implies that there exists an acceptable encoding for data sufficient answers from *CHR^{rp}* into *static CHR^{rp}*.

3.3. Complexity of the encodings

In this section we discuss the complexity of the encodings that we have presented in the previous sections, by taking into account both the size of the translated program and the efficiency of the simulation of the original program.

3.3.1. Encoding static CHR^{rp} into static CHR$^{rp}_2$

Assume that P is a *static CHR^{rp}* program with m rules, that p is the cardinality of the set of all the predicates symbols which occur in the head of a rule in P and that r is the maximal number of constraints appearing in a head of a rule in P. Let us denote by s be the maximum among the quantities m, p and r. From the definition of the encoding it follows immediately that the size of the encoded program $\alpha(P)$ is $O(s^3)$, assuming that equalities and inequalities can be used as built-in constraints. Thus, even though it is larger, the encoded version has still a size which is reasonable when compared with that one of the original program. On the other hand, the execution of the encoded program can be quite inefficient, as shown by the following example.

Example 3. Let us consider the *static CHR^{rp}* program P:

$$1 :: r_1 \ @\emptyset \backslash c(0) \Leftrightarrow \mathit{true}$$
$$2 :: r_2 \ @c(X_1), \ldots, c(X_h) \Rightarrow c(0)$$

By definition $\alpha(P)$ is the following program

$$1 :: \mathit{rule}_{(1,ac)} \ @ \ \mathit{id}(V), ac(X) \Leftrightarrow \mathit{id}(V+1), \mathit{new}_{ac}(V, X)$$
$$2 :: \mathit{rule}_{(2,ac)} \ @ \ ac(X) \Leftrightarrow \mathit{id}(2), \mathit{new}_{ac}(1, X)$$
$$3 :: \mathit{rule}_{(3,2,2)} \ @ \ \mathit{end} \backslash rC[2]_2(\_) \Leftrightarrow \mathit{true}$$
$$\vdots$$
$$3 :: \mathit{rule}_{(3,2,h-1)} \ @ \ \mathit{end} \backslash rC[h-1]_2(\_) \Leftrightarrow \mathit{true}$$
$$3 :: \mathit{rule}_{(4,1,ac)} \ @ \ rA_1(V) \backslash \mathit{new}_{ac}(V', X) \Leftrightarrow V' = V | \mathit{true}$$
$$3 :: \mathit{rule}_{(5,1,2,1)} \ @ \ rA_1(V) \backslash rC[2]_2(\bar{V}', \bar{X}) \Leftrightarrow V \in \bar{V}' | \mathit{true}$$
$$\vdots$$
$$3 :: \mathit{rule}_{(5,1,2,h-1)} \ @ \ rA_1(V) \backslash rC[h-1]_2(\bar{V}', \bar{X}) \Leftrightarrow V \in \bar{V}' | \mathit{true}$$
$$4 :: \mathit{rule}_{(6,1)} \ @ \ rA_1(V) \Leftrightarrow \mathit{true}$$
$$4 :: \mathit{rule}_{(6,2)} \ @ \ rA_2 \Leftrightarrow \mathit{true}$$
$$5 :: \mathit{rule}'_{(2,2)} \ @ \ \mathit{new}_{ac}(V_1, X_1), \mathit{new}_{ac}(V_2, X_2) \Rightarrow V_2 \neq V_1 | rC[2]_2(V_1, V_2, X_1, X_2)$$
$$\vdots$$
$$5 :: \mathit{rule}'_{(2,h)} \ @ \ rC[h-1]_2(\bar{V}_1, \bar{X}_1), \mathit{new}_{ac}(V_2, X_2) \Rightarrow V_2 \notin \bar{V}_1 | rC[h]_2(\bar{V}_1, V_2, \bar{X}_1, X_2)$$
$$7 :: \mathit{rule}_{(7,1)} \ @ \ rC[1]_1(V_1, 0), \mathit{id}(V) \Leftrightarrow rA_1(V_1), \mathit{id}(V)$$
$$8 :: \mathit{rule}_{(7,2)} \ @ \ rC[h]_2(V_1, \ldots, V_h, X_1, \ldots, X_h), \mathit{id}(V) \Leftrightarrow \mathit{new}_{ac}(V, 0), \mathit{id}(V+1), rA_2$$
$$9 :: \mathit{rule}_8 \ @ \ \mathit{id}(V) \Leftrightarrow \mathit{end}$$
$$9 :: \mathit{rule}_9 \ @ \ \mathit{end} \Leftrightarrow \mathit{true}$$

where by convention, for $i \in \{1, 2\}$, $rC[1]_i(V, X) = \mathit{new}_{ac}(V, X)$.

Starting with the goal $c(1), \ldots, c(h)$ the program P fires only once the rule r_2 generating the constraint $c(0)$ that is then immediately removed by rule r_1. On the other hand, in the encoded program $\alpha(P)$ when the equivalent of the constraint $c(0)$ is added rules $\mathit{rule}'(2, N)$ fire, for $N \in [2, h]$. Since all the constraints in the store can match every constraint of rule r_2 and there are $h+1$ of them, the rules $\mathit{rule}'(2, N)$ fire $O(h!)$ times. The simulation of a *static CHR^{rp}* program P can therefore be extremely inefficient, since the simulation of one rule in the original program can require the firing of $O(n!/(n-h)!)$ rules in the encoded program, where n is the number of constraints in the store (which can be very large) and h the maximal number of constraints in a rule head. This is essentially due to the fact that the encoded program needs to consider all the possible propagation rules that can fire after the addition of a constraint and, as we have shown, in the worst case it may involve the firing of a factorial number of rules.

3.3.2. Encoding CHR^{rp} into static CHR^{rp}

Like in the previous case, also when considering the encoding of CHR^{rp} into *static* CHR^{rp} the size of the encoded program $\mathcal{T}(P)$ is polynomial (actually, linear) w.r.t. the size of P. This is due to the fact that the number of new rules added is never greater than the number of constraints in $Head(P)$ or the number of rules of P (multiplied by a constant).

As for the execution time of the encoded program we distinguish two cases: if there are no propagation rules then the encoded program can simulate the firing of a rule in the original program by using $O(D)$ rule firings, where D is the priority of the rule with minimal priority. Indeed the encoded program needs to select, among all the rules that can be fired, which rule has higher priority. This is done by the rules EVALUATE_PRIORITIES that can fire $D-1$ times, in the worst case, because the *highest_priority* constraint cannot be decreased less then the value of the lowest priority, which is $D-1$.

In case propagation rules are used, analogously to the case of previous subsection, it is possible that for simulating a single rule of the original program a factorial number of rules need to be fired in the encoded program. Consider for instance the program of Example 3 with the goal $c(1),\dots,c(h)$. By definition the encoded program $\mathcal{T}(P)$ is the following:

$$1 :: rule_{(1,c)}\ @\ start\backslash id(V), ac(X) \Leftrightarrow id(V+1), new_{ac}(V,X)$$
$$2 :: rule_{(2,c)}\ @\ ac(X) \Rightarrow start, id(0)$$
$$2 :: rule_3\ @\ start \Leftrightarrow highest\_priority(inf)$$
$$3 :: rule_{(4,1)}\ @\ end\backslash instance_1(\_) \Leftrightarrow true$$
$$3 :: rule_{(4,2)}\ @\ end\backslash instance_2(\_) \Leftrightarrow true$$
$$4 :: rule_5\ @\ end \Leftrightarrow true$$
$$6 :: rule_{(7,1)}\ @\ new_{ac}(0,Z)\backslash highest\_priority(inf) \Leftrightarrow highest\_priority(1)$$
$$6 :: rule_{(8,1)}\ @new_{ac}(0,Z)\backslash highest\_priority(P) \Leftrightarrow 1 < P|highest\_priority(1)$$
$$5 :: rule_{(6,2)}@\ new_{ac}(X_1,Z_1),\dots,new_{ac}(X_h,Z_h) \Rightarrow instance_2(Z_1,\dots,Z_h)$$
$$6 :: rule_{(7,2)}\ @\ instance_2(Z_1,\dots,Z_h), new_{ac}(X_1,Z_1),\dots,new_{ac}(X_h,Z_h)\backslash \\ highest\_priority(inf) \Leftrightarrow highest\_priority(2)$$
$$6 :: rule_{(8,2)}\ @\ instance_2(Z_1,\dots,Z_h), new_{ac}(X_1,Z_1),\dots,new_{ac}(X_h,Z_h)\backslash \\ highest\_priority(P) \Leftrightarrow 2 < P|highest\_priority(2)$$
$$7 :: rule_9\ @\ highest\_priority(inf), id(V) \Leftrightarrow end$$
$$8 :: rule_{(10,1)}\ @\ new_{ac}(0,Z), highest\_priority(P), id(V) \Leftrightarrow \\ 1 = P|id(V), highest\_priority(inf)$$
$$8 :: rule_{(10,2)}\ @\ new_{ac}(X_1,Z_1),\dots,new_{ac}(X_h,Z_h)\backslash instance_2(Z_1,\dots,Z_h), \\ highest\_priority(P), id(V) \Leftrightarrow 2 = P|new_{ac}(0,V), \\ id(V+1), highest\_priority(inf)$$

In this case, when the equivalent of the constraint $c(0)$ is added, the encoded program $\mathcal{T}(P)$ starts to fire rules $rule_{(6,2)}$ and, similarly to what happens to rules $rule'(2,N)$ for the $\alpha(P)$ encoding, a factorial number of rules could be fired. Hence, when propagation rules are added the encoded program may be extremely inefficient w.r.t. the original program, since simulating a rule firing of the original program may involve the firing of $O(n!/(n-h)!)$ rules in the encoded program, where n is the number of constraint in the store and h the maximal number of constraint in a rule head.

4. Separation results

In this section we prove that priorities do augment the expressive power of CHR. To do so we prove that there exists no acceptable encoding from *static* CHR^{rp} into CHR^{ω_t}.

In order to prove this separation result we need the following lemma which states a key property of CHR computations under the ω_t semantics. Essentially it says that, given a program P and goal G, if there exists a derivation for G in P which produces a qualified answer (d,K) where d is a built-in constraint, then when considering the goal (d,G) we can perform a derivation in P, which is essentially the same as the previous one, with the only exception of a Solve transition step (in order to evaluate the constraint d). Hence it is easy to observe that such a new computation for (d,G) in P terminates producing the same qualified answer (d,K).

The proof of the following Lemma is then immediate.

Lemma 1. *Let P be a CHR^{ω_t} program and let G be a goal. Assume that G in P has the qualified answer (d,K). Then the goal (d,G) has the same qualified answer (d,K) in P.*

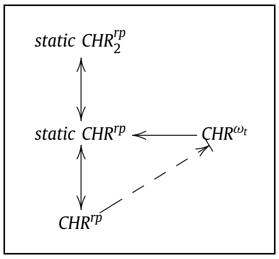

Fig. 3. Graphical summary: $\dashrightarrow$: absence of an acceptable encoding $\rightarrow$: presence of an acceptable encoding.

Lemma 1 is not true anymore if we consider *CHR*^{rp} programs. Indeed if we consider the program P consisting of the rules

$$1 :: h(X) \Leftrightarrow X = yes | false$$

$$2 :: h(X) \Leftrightarrow X = yes$$

then the goal $h(X)$ has the qualified answer $X = yes$ in P, while the goal $X = yes, h(X)$ has no qualified answer in P. With the help of the previous lemma we can now prove our main separation result.

Theorem 3. *There exists no acceptable encoding for data sufficient answers from CHR^{rp} into CHR$^{\omega_t}$.*

Proof. The proof is by contradiction. Consider the following program P in *CHR*^{rp}

$$1 :: h(X) \Leftrightarrow X = yes | false$$

$$2 :: h(X) \Leftrightarrow X = yes$$

and assume that $(\gamma, \mathcal{INP}, \mathcal{OUT})$ is an acceptable encoding for data sufficient answers from *CHR*^{rp} into *CHR*$^{\omega_t}$.

Let G be the goal $h(X)$. Then $\mathcal{SA}_P(G) = \{X = yes\}$. Since the goal $h(X)$ has the data sufficient answer $X = yes$ in the program P and since the encoding preserves data sufficient answers, $\mathcal{QA}_{\gamma(P)}(\mathcal{INP}(h(X)))$ contains a qualified answer S such that $\mathcal{OUT}(S) = (X = yes)$. Moreover, since the output decoding function is such that the built-ins appearing in the answer are left unchanged, we have that S is of the form $(X = yes, K)$, where K is a (possibly empty) multiset of CHR constraints.

Then since the goal encoding function is such that the built-ins present in the goal are left unchanged $\mathcal{INP}(X = yes, h(X)) = (X = yes, \mathcal{INP}(h(X)))$ and therefore from previous Lemma 1, it follows that the program $\gamma(P)$ with the goal $\mathcal{INP}(X = yes, h(X))$ has the qualified answer S.

However $(X = yes, h(X))$ has no data sufficient answer in the original program P. This contradicts the fact that $(\gamma, \mathcal{INP}, \mathcal{OUT})$ is an acceptable encoding for data sufficient answers from *CHR*^{rp} into *CHR*$^{\omega_t}$, thus concluding the proof. $\square$

Since the existence of an acceptable encoding implies the existence of an acceptable encoding for data sufficient answers we have the following immediate corollary:

Corollary 2. *There exists no acceptable encoding from CHR^{rp} into CHR$^{\omega_t}$.*

5. Conclusions

We have studied the expressive power of CHR with priorities and we have shown that, differently from the case of standard CHR, allowing more than two atoms in the head of rules does not augment the expressive power of the language. We have also proved that dynamic priorities do not increase the expressive power w.r.t. static ones. These results are proved by providing translations from *static CHR*^{rp} into *static CHR*$_2^{rp}$ and from *CHR*^{rp} into *static CHR*^{rp} which preserve the standard observables of CHR computations (qualified answers). (See Fig. 3.)

Actually these translations would allow to prove stronger results, since each computational step of the original program is simulated precisely by several steps of the translated program. This means that if for example the rule bodies would have non-pure observable side-effects, the encoding would still correctly capture them.[4] We have also discussed the complexity

[4] This observation was made by a reviewer of this paper.

of our encodings, showing that the translated programs have a size which is polynomial (in one case linear) in the size of the original one, while their execution can be quite inefficient in time when propagation rules are allowed.

Concerning negative (i.e. separation) results instead, we have proved that, when considering the theoretical semantics, there exists no acceptable encoding of CHR with (static) priorities into standard CHR. This means that, even though both languages are Turing powerful, priorities augment the expressive power of the language in a quite reasonable sense, as discussed in the introduction.

This paper was inspired by the work of De Koninck et al. [3], where the *CHR^rp* language was introduced and where it was shown that *static CHR^rp* programs can be translated into CHR with the refined semantics. However that paper did not provide the formal results that we have shown here.

Among the other few papers which consider the expressive power of CHR a quite relevant one is [14], where the authors show that it is possible to implement any algorithm in CHR in an efficient way, i.e. with the best known time and space complexity. This result is obtained by introducing a new model of computation, called the CHR machine, and comparing it with the well-known Turing machine and RAM machine models. Earlier works by Frühwirth [8,9] studied the time complexity of simplification rules for naive implementations of CHR. In this approach an upper bound on the derivation length, combined with a worst case estimate of (the number and cost of) rule application attempts, allows to obtain an upper bound of the time complexity. The aim of all these works is different from ours, even though they can be used to state that, in terms of classical computation theory, *CHR^rp* is equivalent to CHR.

Another paper which studies the expressive power of CHR is [13], where the author shows that several subclasses of CHR are still Turing-complete, while single-headed CHR without host language and propositional abstract CHR are not Turing-complete. Recently these results have been further extended in [5].

Our notion of acceptable encoding has been recently used in [1] to justify a source-to-source transformation.

When moving to the more general field of concurrent languages one can find several works related to the present one. In particular, concerning priorities, Versari et al. [17] show that the presence of priorities in process algebras does augment the expressive power. More precisely the authors show, among other things, that a finite fragment of asynchronous CCS with (global) priority cannot be encoded into π-calculus nor in the broadcast based $b-\pi$ calculus. This result is related to our separation result for *CHR^rp* and CHR, even though the formal setting is completely different.

More generally, often in process calculi and in distributed systems separation results are obtained by showing that a problem can be solved in a language and not in another one (under some additional hypothesis, similar to those used here). For example, in [11] the author proves that there exists no *reasonable* encoding from the π-calculus to the asynchronous π-calculus by showing that the symmetric leader election problem has no solution in the asynchronous version of the π-calculus. A survey on separation results based on this problem can be found [18].

Our work could be continued by investigating some conjectures. In particular, the priorities seem related to negation as absence [16] in the sense that, as mentioned in Section 4, by using priorities one can check the absence of information. Therefore it seems that one can encode negation as absence in *CHR^rp*.

Appendix A. α Encoding without equalities or inequalities

The rules in the α encoding can be written also without the support of a constraint theory allowing equality and inequality as built-ins. In fact, rules $rule'_{(i,N)}$ can be translated into rules having inequalities in their guards. These rules have a structure similar to the following rule:

$$p :: c_1(V, \bar{X}), c_2(V_1, \ldots, V_n, \bar{Y}) \Rightarrow V \neq V_1, \ldots, V \neq V_n | c_3(V, V_1, \ldots, V_n, \bar{X}, \bar{Y})$$

Since we know that $V, V_1, \ldots, V_n$ will always be matched with ground terms $t, t_1, \ldots, t_n$, such that $t_1, \ldots, t_n$ are all different from each other, we can replace the previous rule with the following rules:

$$\begin{gathered} p_1 :: eq(\bar{Z}) \backslash c_3(\bar{Z}) \Leftrightarrow true \\ p_2 :: c_1(V, \bar{X}), c_2(V_1, \ldots, V_n, \bar{Y}) \Rightarrow V = V_1 | eq(V, V_1, \ldots, V_n, \bar{X}, \bar{Y}) \\ \ldots \\ p_2 :: c_1(V, \bar{X}), c_2(V_1, \ldots, V_n, \bar{Y}) \Rightarrow V = V_n | eq(V, V_1, \ldots, V_n, \bar{X}, \bar{Y}) \\ p_3 :: c_1(V, \bar{X}), c_2(V_1, \ldots, V_n, \bar{Y}) \Rightarrow c_3(V, V_1, \ldots, V_n, \bar{X}, \bar{Y}) \end{gathered}$$

where p_1, p_2, p_3 are priorities s.t. $p_1 < p_2 < p_3$ and *eq* is a new constraint. Thus we have removed inequalities. Equalities instead can be removed by simply changing the name of terms in the head of the rules. For instance the equality $X = Y$ in a rule like

$$k_1(X), k_2(Y) \Leftrightarrow X = Y | C$$

can be removed replacing the previous rule with the following one

$$k_1(X), k_2(X) \Leftrightarrow C$$

Appendix B. Theorem 1

Theorem 1. *The triple $(\alpha, \mathcal{INP}, \mathcal{OUT})$ provides an acceptable encoding from static CHR^{rp} into static CHR$_2^{rp}$.*

Proof. By definition, we have to prove that for all *static CHR^{rp}* programs P and goals G,

$$\mathcal{QA}_P(G) = \mathcal{OUT}\big(\mathcal{QA}_{\alpha(P)}\big(\mathcal{INP}(G)\big)\big)$$

holds.

By construction, the functions $\mathcal{INP}$ and $\mathcal{OUT}$ are compositional and defined as:

$$\mathcal{INP}\big(b(\bar{t})\big) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ ab(\bar{t}) & \text{otherwise} \end{cases}$$

$$\mathcal{OUT}\big(b(\bar{t})\big) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ k(\bar{t}') & \text{if } b(\bar{t}) = new_{ak}(V, \bar{t}') \\ k(\bar{t}) & \text{if } b(\bar{t}) = ak(\bar{t}) \end{cases}$$

Let P be a *static CHR^{rp}* program and let G be a goal. From definition of $\mathcal{INP}$ we have that the predicate symbols *id*, *end*, $rC[N]_i$, rA_i, new_k (with $k \in \mathcal{INP}(Head(P))$) cannot be in the encoded goal $\mathcal{INP}(G)$.

Therefore if $G = \emptyset$ or G does not contain predicate symbols that are in $Head(P)$ we have that $\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\alpha(P)}(\mathcal{INP}(G)))$ since no rule from both the two programs can be applied. If however the goal G contains a predicate symbol in $Head(P)$ (and therefore $\mathcal{INP}(G)$ contains a predicate symbol in $\mathcal{INP}(Head(P))$) then $rule_{(2,k)} \in \alpha(P)$ is fired first. At this point, each constraint $k(\bar{t})$ in G, such that $k \in Head(P)$, is transformed by rule $rule_{(1,k)}$ into the constraint $new_{\mathcal{INP}(k)}(n, \bar{t})$, where n is a unique identifier (intuitively this identifier can be considered as the identifier assigned to the original constraint by the Introduce transition step). Let us define the mapping between the original constraint $k(\bar{t})$ with the corresponding n identifier of the $new_{\mathcal{INP}(k)}(n, \bar{t})$ constraint as φ.

After this phase we obtain a new goal G' in $\alpha(P)$ and the rules $rule_{(2,k)}$ and $rule_{(1,k)}$ are no longer used in this derivation in $\alpha(P)$. Since there is no *end* or rA_i predicate symbol in G', the next rules that are applied in the derivation in $\alpha(P)$ are rules $rule'_{(i,N)}$. By definition of these rules a constraint $rC[N]_i(V_1, \dots, V_N, \bar{t})$ is generated if in the original program the constraints $\varphi^{-1}(V_1), \dots, \varphi^{-1}(V_N)$ in G can be used as a match for the application of the rule $rule_i$ in P. Thus a constraint $rC[r_i]_i(\bar{V}, \bar{t})$ is created for every possible match of constraints that can fire rule $rule_i$.

When all the possible $rule'_{(i,N)}$ have fired there are two possibilities:

1. If in the original program a rule can fire then at least one rule $rule_{(7,i)}$ fires. The firing of this rule corresponds to the firing of a rule $rule_i$ in the original program. For every constraint $k(\bar{t})$ in the body of the original rule a new $new_{\mathcal{INP}(k)}(V, \bar{t})$ constraint is added to the store of the derivation in the encoded program, with its new unique identifier V. This rule also adds to the store the constraint $rA_i(\bar{V})$ where $\bar{V}$ are the identifiers $\varphi(k(\bar{t}))$ of all the constraints $k(\bar{t})$ that are removed from the store by the application of $rule_i$ in the original program P. The removal of this constraints in the encoded program is done by rules $rule_{(4,i,k)}$ that are eventually fired immediately after rule $rule_{(7,i)}$. Rules $rule_{(5,j,i,k)}$ are then fired for removing all the constraints $rC[N]_i$ that have no more sense to exist since one of the constraints identified by their arguments has been removed. After that, the constraint $rA_i(\bar{V})$ is no longer useful and it is removed by rule $rule_{(6,i)}$. When the constraint $rA_i(\bar{V})$ is removed other rules $rule'_{(i,N)}$ can fired (new $new_{\mathcal{INP}(k)}(\_)$ constraints have potentially been added to the store by $rule_{(7,i)}$) repeating the cycle.
2. If in the original program no rule can fire then no rule $rule_{(7,i)}$ in $\alpha(P)$ can fire and therefore $rule_8$ fires. This removes the constraint *id* and adds the constraint *end* that triggers the rules $rule_{(3,i,N)}$. These rules remove all the constraints $rC[N]_i$ and when all these constraints are removed the *end* constraint is removed too by $rule_9$. After the firing of this rule, no rule of the program can fire anymore.

For every rule $rule_i$ that can fire in the original program there is a corresponding rule $rule_{(7,i)}$ that can fire in the encoded program. Moreover for every CHR constraint $k(\bar{t})$ in every configuration during the execution of the goal G in P we have two possibilities. If $k \notin Head(P)$ and $k(\bar{t})$ is in the initial goal G, then there is a $\mathcal{INP}(k)(\bar{t})$ constraint in the correspondent configuration during the execution of $\mathcal{INP}(G)$ in $\alpha(P)$. If $k \in Head(P)$ or $k(\bar{t})$ is introduced by an Apply transition step, then there is a $new_{\mathcal{INP}(k)}(V, \bar{t})$ constraint in the correspondent configuration during the execution of $\mathcal{INP}(G)$ in $\alpha(P)$. The built-in constraints are not modified and are processed in the same way by both the two programs.

When the encoded program terminates no *id*, *end*, $rC[N]_i$, rA_i are in the store.[5] Hence applying the decoding function to the qualified answer of the encoded program produces the equivalent qualified answer of the original program. □

[5] Technically speaking rules $rule_{(3,i,N)}$, $rule_8$ and $rule_9$ are not needed because the constraints can be removed using the decoding function. We chose to add them to exploit the same encoding also for Theorem 3.

Appendix C. β Encoding

If data sufficient answers are considered it is possible to strengthen Theorem 1 by requiring that the goal encoding and the output decoding functions are the identity functions. To do so it is possible to use the α encoding as a base for a new program encoding for data sufficient answers exploiting the fact that when a fresh constraint for a program P is in a goal then the program has no data sufficient answers for that goal.

Below we exploit this idea and we first define a program translation $\beta(P,q)$ that, given a *static CHR*^{rp} program P and a predicate symbol q produces a modified program P' which has the same data sufficient answers as P for every goal that does not contain the predicate symbol q, and produces a failure otherwise.[6]

Let us then consider a *static CHR*^{rp} program P composed from m rules

$$p_i :: rule_i \ @ \ H_i \backslash H'_i \Leftrightarrow G_i | B_i$$

where $1 \leqslant p_i \leqslant p_{max}$. W.l.o.g., we can assume that *start* and *init* are not contained in *Head*(P). Moreover, let f be an injective function that maps predicate symbols into predicate symbols which are not in *Pred*$(P) \cup \{start, init, q\}$. f can be extended to a multiset of constraints in the obvious way.

The transformation $\beta(P,q)$ produces the following program

$$1 :: rule_{m+1} \ @ \ start, q(\_) \Leftrightarrow false$$

for every predicate name $k \in Head(P)$

$$1 :: rule_{(m+2,k)} \ @ \ start, \ f(k(\_)) \Leftrightarrow false$$

$$1 :: rule_{m+3} \ @ \ start, init \Leftrightarrow false$$

$$2 :: rule_{m+4} \ @ \ start \Leftrightarrow init$$

for every predicate name $k, k' \in Head(P)$

$$3 :: rule_{(m+5,k)} \ @ \ k(\_) \Rightarrow start$$

$$3 :: rule_{(m+6,k,k')} \ @ \ k(\_) \backslash k'(\bar{Y}) \Leftrightarrow f(k'(\bar{Y}))$$

for every predicate name $k \in Head(P)$

$$4 :: rule_{(m+7,k)} \ @ \ k(X) \Rightarrow f(k(\bar{X}))$$

for every $i \in \{1, \ldots, m\}$

$$4 + p_i :: rule'_i \ @ \ f(H_i) \backslash f(H'_i), \Leftrightarrow G_i | f(B_i), init$$

$$5 + p_{max} :: rule_{(m+8)} \ @ \ init, init \Leftrightarrow init$$

for every predicate name $k \in Head(P)$

$$6 + p_{max} :: rule_{(m+9,k)} \ @ \ k(\_), init \Leftrightarrow true$$

The following lemma shows that indeed the transformed program has the behavior that we have described before.

Lemma 2. *Let P be a static CHR^{rp} program and let q be a predicate symbol. For every goal G, if G does not contain the predicate symbol q then $\mathcal{SA}_P(G) = \mathcal{SA}_{\beta(P,q)}(G)$, $\mathcal{SA}_{\beta(P,q)}(G) = \emptyset$ otherwise.*

Proof. By our assumptions, *start* and *init* are not contained in *Head*(P). Moreover, by construction, f is a function that maps predicate symbols into fresh predicate symbols (i.e. not in *Pred*$(P) \cup \{start, init, q\}$).

The proof is by cases on the form of the goal G.

If $G = \emptyset$ or G does not contain predicate symbols that are in *Head*(P) we have that $\mathcal{SA}_P(G) = \emptyset$. Moreover since in $\beta(P,q)$ there is no rule which produces an atom of the form $k(\bar{t})$, with $k \in Head(P)$, we have that $rule_{(m+9,k)}$ cannot be used and therefore $\mathcal{SA}_{\beta(P,q)}(G) = \emptyset$.

Now, let us to assume that the goal G contains a predicate symbol in *Head*(P). We have the following cases.

$(G = start, G')$ In this case, since by our assumptions $start \notin Head(P)$ we have that $\mathcal{SA}_P(G) = \emptyset$. Moreover we have the following possibilities

1. $(init \in G'$ or $q(\_) \in G'$ or $f(k)(\_) \in G'$, with $k \in Head(P))$. In this case

$$\langle G, \emptyset, true, \emptyset\rangle_1 \rightarrow^{\omega_p}_{\beta(P,q)}{}^{*} \langle \emptyset, G'', false, T\rangle_n$$

by using one of the three clauses with priority 1.

[6] Note that we are not requiring that the presence of a constraint of the form $q(\bar{t})$ always leads to a failure. We allow for instance the use of $q(\bar{t})$ during the execution. The program fails only if a constraint of the form $q(\bar{t})$ is in the original goal.

Therefore $\mathcal{SA}_{\beta(P,q)}(G) = \mathcal{SA}_P(G) = \emptyset$.

2. ($init \notin G'$, $q(\_) \notin G'$, $f(k)(\_) \notin G'$ with $k \in Head(P)$, and $start \in G'$). In this case

$$\langle G, \emptyset, true, \emptyset\rangle_1 \xrightarrow{\omega_p}{}^{*}_{\beta(P,q)} \langle \emptyset, (G'', start\#l, start\#p), B, T\rangle_k \xrightarrow{\omega_p}_{\beta(P,q)}$$
$$\langle \emptyset, (G'', start\#l, init\#n), B, T'\rangle_{n+1} \xrightarrow{\omega_p}{}^{*}_{\beta(P,q)} \langle \emptyset, G'', false, T''\rangle_{n+1}$$

by using in the order the rules $rule_{m+4}$ and $rule_{m+3}$ and therefore $\mathcal{SA}_{\beta(P,q)}(G) = \mathcal{SA}_P(G) = \emptyset$.

3. ($G' = k_1(\bar{t}_1), \ldots, k_r(\bar{t}_r)$, with $k_i \in Head(P)$, for $i = 1, \ldots, r$). In this case, after some Solve and Introduce transition steps and an Apply transition step we have that

$$\langle G, \emptyset, true, \emptyset\rangle_1 \xrightarrow{\omega_p}{}^{*}_{\beta(P,q)} \langle \emptyset, G', B, T\rangle_n$$

where
   - ∘ either G' is of the form $(G'', start\#l, start\#p)$ if the Apply transition step uses $rule_{(m+5,k)}$
   - ∘ or G' is of the form $(G'', start\#l, f(k'(\bar{t}'))\#p)$ if the Apply transition step uses a rule $rule_{(m+6,k,k')}$.

   By using the same arguments of the cases 1 and 2, we have that $\mathcal{SA}_{\beta(P,q)}(G) = \mathcal{SA}_P(G) = \emptyset$.

($start \notin G$) We have two further cases.

1. (G contains an atom of the form $init$ or $f(k(\bar{t}))$ with $k \in Head(P)$). Since by our hypothesis $init, f(k) \notin Head(P)$ we have that $\mathcal{SA}_P(G) = \emptyset$ and since G contains at least an atom of the form $k(\bar{t})$, with $k \in Head(P)$, it is easy to check that

$$\langle G, \emptyset, true, \emptyset\rangle_1 \xrightarrow{\omega_p}{}^{*}_{\beta(P,q)} \langle \emptyset, (G', start\#p), B, T\rangle_n$$

by using some Apply transition steps with $rule_{(m+6,k,k')}$ and then an Apply transition step with $rule_{(m+5,k)}$, where G' contains an atom of the form $init$ or $f(k(\bar{t}))$ with $k \in Head(P)$. In this case, analogously to point 1 of the case $(G = start, G')$, we have that the derivation ends in a failed configuration and then $\mathcal{SA}_{\beta(P,q)}(G) = \mathcal{SA}_P(G) = \emptyset$.

2. (G contains an atom of the form $q(\bar{t})$). Then, analogously to the previous point, we have that the derivation of G in $\beta(P,q)$ ends in a failed configuration and then $\mathcal{SA}_{\beta(P,q)}(G) = \emptyset$.

3. ($G = k_1(\bar{t}_1), \ldots, k_n(\bar{t}_n)$). Let us consider a derivation δ for G in $\beta(P,q)$. We distinguish two cases:

   (The first Apply transition step uses a rule $rule_{(m+6,k,k')}$). In this case, analogously to point 3 of the case ($start \in G$), we have that δ ends in a failed configuration.

   (The first Apply transition step uses a rule $rule_{(m+5,k)}$). W.l.o.g., we can assume that $rule_{(m+5,k_l)}$ rewrites an atom of the form $k_l(\bar{t}_l)$. Then we have that

$$\delta = \langle G, \emptyset, true, \emptyset\rangle_1 \xrightarrow{\omega_p}{}^{*}_{\beta(P,q)} \langle \emptyset, G', B, \emptyset\rangle_n \xrightarrow{\omega_p}_{\beta(P,q)}$$
$$\langle \emptyset, (G', start\#n), B, \{[s, rule_{(m+5,k_l)}]\}\rangle_{n+1}$$
$$\langle \emptyset, (G', init\#n+1), B, \{[s, rule_{(m+5,k_l)}], [n, rule_{m+4}]\}\rangle_{n+2} \cdot \delta'$$

   Now, we have two further possibilities.

   3.1. There exists an atom in G', which is rewritten by using a clause $rule_{(m+5,j)}$.
   In this case

$$\delta' = \langle \emptyset, (G', init\#n+1), B, \{[s, rule_{(m+5,k_l)}], [n, rule_{m+4}]\}\rangle_{n+2}$$
$$\xrightarrow{\omega_p}{}^{*}_{\beta(P,q)} \langle \emptyset, (G'', init\#n+1, start\#n'), B', T'\rangle_{n'+1}$$
$$\xrightarrow{\omega_p}{}^{*}_{\beta(P,q)} \langle \emptyset, G_1, false, T'\rangle_{n'+1}$$

   where the last Apply transition step uses either $rule_{(m+2,k)}$ or $rule_{m+3}$.

   3.2. There exists no atom in G', which is rewritten by using a clause $rule_{(m+5,k)}$.
   In this case

$$\langle \emptyset, (G', init\#n+1), B, \{[s, rule_{(m+5,k_l)}], [n, rule_{m+4}]\}\rangle_{n+2}$$
$$\xrightarrow{\omega_p}{}^{*}_{\beta(P,q)} \langle \emptyset, (G'', init\#n+1, k_l(\bar{t}_l)\#s, B', T'\rangle_{n''}$$

   where $\mathtt{chr}(G'') = f(k_1(\bar{t}_1)), \ldots, f(k_n(\bar{t}_n))$, all the Apply transition steps except the last one use one of the rules $rule_{(m+6,k,k')}$, $[s, rule_{(m+5,k_l)}] \in T'$ and the last Apply transition step rewrites the atom $k_l(\bar{t}_l)\#s$ by using the rule $rule_{(m+7,k_l)}$ (and therefore $[s, rule_{(m+7,k_l)}] \in T'$). From this point the only applicable rules are $rule'_i$, $rule_{(m+8,k)}$ and $rule_{(m+9,k)}$.

   Then the proof is immediate by previous results and by definition of $rule_i$, $rule'_i$, $rule_{(m+8,k)}$ and $rule_{(m+9,k)}$. □

Now let us denote with β' the extensions of β to a list of predicate symbols ($\beta'(P,[]) = P$ and $\beta'(P,[X|XS]) = \beta(\beta'(P,XS),X)$).

Suppose that $New\_Symbols(P)$ is the list of the new predicate symbols introduced by $\alpha(P)$ (namely *id*, *end*, $rC[N]_i$, rA_i, new_{ak}) and w.l.o.g. suppose that these predicate symbols are fresh in P.

Using the lemma we can prove that if data sufficient are considered it is possible to obtain an acceptable encoding where the goal encoding and the output decoding functions are the identity functions.

Theorem 3. *The triple $(\beta'(\alpha(P), New\_Symbols(P)), id, id)$, where id is the identity function and α is defined as before, provides an acceptable encoding for data sufficient answers from static CHR^{rp} into static CHR$^{rp}_2$.*

Proof. The proof derives from Lemma 2 using the program encoding of Theorem 1. Indeed given a program P w.l.o.g. we can assume that *id*, *end*, $rC[N]_i$, rA_i and new_{ak} (with $k \in Head(P)$) are not contained in $Head(P)$. Therefore for every goal G containing at least one of them, we have that

$$\mathcal{SA}_P(G) = \emptyset.$$

By using the same arguments of Theorem 1, for each goal G s.t. no predicate symbol in $New\_Symbols(P)$ is in G we have that $\mathcal{SA}_P(G) = \mathcal{SA}_{\alpha(P)}(G)$. Moreover, by construction, $\alpha(P) \in$ *static CHR*$^{rp}_2$. By Lemma 2 for every goal G, if G does not contain the predicate symbols in $New\_Symbols(P)$ then $\mathcal{SA}_P(G) = \mathcal{SA}_{\beta'(P,New\_Symbols(P))}(G)$, $\mathcal{SA}_{\beta'(P,New\_Symbols(P))}(G) = \emptyset$ otherwise. Therefore we have that for each goal G, $\mathcal{SA}_{\beta'(\alpha(P),New\_Symbols(P))}(G) = \mathcal{SA}_P(G)$. Moreover, since $\alpha(P) \in$ *static CHR*$^{rp}_2$, by definition of $\beta'()$ we have that $\beta'(\alpha(P), New\_Symbols(P)) \in$ *static CHR*$^{rp}_2$ and then the thesis. □

It is worth noting that Theorem 3 does not hold when the traditional semantics is considered, as shown in [5].

Appendix D. Theorem 2

Theorem 2. *The triple $(\mathcal{T}, \mathcal{INP}, \mathcal{OUT})$ provides an acceptable encoding between CHR^{rp} and static CHR^{rp}.*

Proof. By definition, we have to prove that for all *CHR*^{rp} programs P and goals G,

$$\mathcal{QA}_P(G) = \mathcal{OUT}\big(\mathcal{QA}_{\mathcal{T}(P)}\big(\mathcal{INP}(G)\big)\big)$$

holds.

By construction, the functions $\mathcal{INP}$ and $\mathcal{OUT}$ are compositional and defined as:

$$\mathcal{INP}\big(b(\bar{t})\big) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint,} \\ ab(\bar{t}) & \text{otherwise} \end{cases}$$

$$\mathcal{OUT}\big(b(\bar{t})\big) = \begin{cases} b(\bar{t}) & \text{if } b(\bar{t}) \text{ is a built-in constraint} \\ k(\bar{t}') & \text{if } b(\bar{t}) = new_{ak}(V, \bar{t}') \\ k(\bar{t}) & \text{if } b(\bar{t}) = ak(\bar{t}) \end{cases}$$

Let P be a *CHR*^{rp} program and let G be a goal.

For the definition of $\mathcal{INP}$ we have that the constraints *start*, *id*, *end*, $instance_i$, *highest_priority*, new_{ak} (where $ak \in \mathcal{INP}(Head(P))$) cannot be in the encoded goal $\mathcal{INP}(G)$.

Therefore if $G = \emptyset$ or G does not contain constraints that are in $Head(P)$ we have that $\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\mathcal{T}(P)}(\mathcal{INP}(G)))$ since no rule from both the two programs P and $\mathcal{T}(P)$ can be applied. If however the goal G contains a constraint in $Head(P)$ then $rule_{(2,k)}$ is fired first. At this point each constraint $ak(\bar{t})$ in $\mathcal{INP}(G)$ (corresponding to a constraint $k(\bar{t})$ in G) such that $k \in Head(P)$ is transformed by rules $rule_{(1,k)}$ into the constraint $new_{ak}(V,\bar{t})$ where V is a unique identifier (intuitively this identifier can be considered as the identifier assigned to the original constraint by the Introduce transition step). Let us define the mapping between the original constraint $k(\bar{t}) \in G$ with the corresponding V identifier of the new_{ak} constraint as φ.

After this phase the rules $rule_{(2,k)}$ and $rule_{(1,k)}$ are no longer used in a derivation in $\mathcal{T}(P)$ and the configuration S is generated. Since there is no *start*, *end* or $instance_i$ constraint in S (they cannot be in the encoded goal $\mathcal{INP}(G)$ due to the goal encoding function) the next rules that are applied in $\mathcal{T}(P)$ are rules in `EVALUATE_PRIORITIES`(i). By definition of these rules, if in the original program P it is possible to fire the j-th rule starting from G, the constraint $highest\_priority(p_j)$ can added to the CHR store of S in $\mathcal{T}(P)$ after all the possible rules in `EVALUATE_PRIORITIES`(i) have fired.

Note that, after all the possible rules in `EVALUATE_PRIORITIES`(i) (for $i = 1,\dots,n$) have fired at most a constraint $highest\_priority(p_j)$ is present in the constraint store.

When all the possible EVALUATE_PRIORITIES(i) (for $i = 1, \dots, n$) have fired there are two possibilities:

1. if in the original program a rule can fire then at least one rule ACTIVATE_RULE(i) (for $i = 1, \dots, n$) fires. The firing of the rule $rule_{(10,j)}$ in $\mathcal{T}(P)$ corresponds to the firing of the j-th rule in the original program P. Moreover, the application of the rule $rule_{(10,j)}$ in $\mathcal{T}(P)$ uses the atoms $p_1(V_1, \bar{t}_1), \dots, p_m(V_m, \bar{t}_m), highest\_priority(p_j), id(l)$ if and only if in the original program $rule_j$ in P can fire by using the atoms $\varphi^{-1}(V_1), \dots, \varphi^{-1}(V_m)$.
   For every constraint $k(\bar{t})$ in the body of the original rule a $new_{ak}(V, \bar{t})$ constraint is added with its new unique identifier V. This rule also adds to the store the constraint $highest\_priority(inf)$ and then the computation starts from EVALUATE_PRIORITIES(i) (for $i = 1, \dots, n$), repeating the cycle;
2. if in the original program no rule can fire then no rule EVALUATE_PRIORITIES(i) (for $i = 1, \dots, n$) can fire and therefore $rule_9$ fires. This removes the constraints $highest\_priority(inf)$ and $id(V)$ from the constraint store. It also adds the constraint *end* that triggers the rules $rule_{(4,i)}$ (for $i = 1, \dots, n$). These rules remove all the constraints $instance_i(\bar{V})$ and when all these constraints are removed the *end* constraint is removed too by $rule_5$. After the firing of this rule no rule of the program can fire anymore.

For every rule $rule_i$ that can fire in the original program P there is a corresponding rule $rule_{(10,i)}$ that can fire in the encoded program $\mathcal{T}(P)$. Moreover for every CHR constraint $k(\bar{t})$ in every configuration during the execution of the goal G in P we have two possibilities. If $k \in Head(P)$ or $k(\bar{t})$ is introduced by an Apply transition step then there is a $new_{ak}(V, \bar{t})$ constraint in the correspondent configuration during the execution of $\mathcal{INP}(G)$ in $\mathcal{T}(P)$. If $k \notin Head(P)$ and $k(\bar{t})$ is in the initial goal G then there is a $ak(\bar{t})$ constraint in the correspondent configuration during the execution of $\mathcal{INP}(G)$ in $\mathcal{T}(P)$. The built-in constraints are not modified and are processed in the same way by both the two programs. When the execution of $\mathcal{INP}(G)$ in $\mathcal{T}(P)$ terminates no *id*, *start*, *highest_priority*, *end*, $instance_i$ are in the store.[7] Hence applying the decoding function to the qualified answer of the encoded program produces the equivalent qualified answer of the original program. $\square$

References

[1] H. Betz, F. Raiser, T.W. Frühwirth, A complete and terminating execution model for constraint handling rules, TPLP 10 (4–6) (2010) 597–610.
[2] F.S.D. Boer, C. Palamidessi, Embedding as a tool for language comparison, Inform. and Comput. 108 (1) (1994) 128–157.
[3] L. De Koninck, T. Schrijvers, B. Demoen, User-definable rule priorities for CHR, in: M. Leuschel, A. Podelski (Eds.), PPDP, ACM, 2007, pp. 25–36.
[4] C. Di Giusto, M. Gabbrielli, M.C. Meo, Expressiveness of multiple heads in CHR, in: M. Nielsen, A. Kucera, P.B. Miltersen, C. Palamidessi, P. Tuma, F.D. Valencia (Eds.), SOFSEM, in: Lecture Notes in Computer Science, vol. 5404, Springer, 2009, pp. 205–216.
[5] C. Di Giusto, M. Gabbrielli, M.C. Meo, On the expressive power of multiple heads in CHR, ACM Trans. Comput. Log. 13 (1) (2012) 6.
[6] G.J. Duck, P.J. Stuckey, M.J.G. de la Banda, C. Holzbaur, The refined operational semantics of constraint handling rules, in: B. Demoen, V. Lifschitz (Eds.), ICLP, in: Lecture Notes in Computer Science, vol. 3132, Springer, 2004, pp. 90–104.
[7] T.W. Frühwirth, Theory and practice of constraint handling rules, J. Log. Program. 37 (1–3) (1998) 95–138.
[8] T.W. Frühwirth, As time goes by II: More automatic complexity analysis of concurrent rule programs, Electron. Notes Theor. Comput. Sci. 59 (3) (2001).
[9] T.W. Frühwirth, As time goes by: Automatic complexity analysis of simplification rules, in: KR 02, 2002.
[10] M. Gabbrielli, J. Mauro, M.C. Meo, On the expressive power of priorities in CHR, in: A. Porto, F.J. López-Fraguas (Eds.), PPDP, ACM, 2009, pp. 267–276.
[11] C. Palamidessi, Comparing the expressive power of the synchronous and asynchronous *pi*-calculi, Math. Structures Comput. Sci. 13 (5) (2003) 685–719.
[12] E.Y. Shapiro, The family of concurrent logic programming languages, ACM Comput. Surv. 21 (3) (1989) 413–510.
[13] J. Sneyers, Turing-complete subclasses of CHR, in: M.G. de la Banda, E. Pontelli (Eds.), ICLP, in: Lecture Notes in Computer Science, vol. 5366, Springer, 2008, pp. 759–763.
[14] J. Sneyers, T. Schrijvers, B. Demoen, The computational power and complexity of constraint handling rules, ACM Trans. Program. Lang. Syst. 31 (2) (2009).
[15] F.W. Vaandrager, Expressive results for process algebras, in: Proceedings of the REX Workshop on Sematics: Foundations and Applications, Springer-Verlag, London, UK, 1993, pp. 609–638.
[16] P. Van Weert, J. Sneyers, T. Schrijvers, B. Demoen, Extending CHR with negation as absence, in: T. Schrijvers, T. Frühwirth (Eds.), Proceedings of the Third Workshop on Constraint Handling Rules, June 2006, pp. 125–139.
[17] C. Versari, N. Busi, R. Gorrieri, On the expressive power of global and local priority in process calculi, in: L. Caires, V.T. Vasconcelos (Eds.), CONCUR, in: Lecture Notes in Computer Science, vol. 4703, Springer, 2007, pp. 241–255.
[18] M.G. Vigliotti, I. Phillips, C. Palamidessi, Tutorial on separation results in process calculi via leader election problems, Theor. Comput. Sci. 388 (1–3) (2007) 267–289.

[7] Technically speaking rules $rule_{(4,i)}$, $rule_5$ and $rule_9$ are not needed because the constraints can be removed using the decoding function.