Expressive power of Constraint Handling Rules extensions and fragments

Jacopo Mauro
University of Bologna / INRIA
jmauro@cs.unibo.it

Abstract

Constraints can be used in concurrency theory to increase the conciseness and the expressive power of concurrent languages from a pragmatic point of view. In this work we study the expressive power of a concurrent language, namely Constraint Handling Rules, that supports constraints as a primitive construct. We show what features of this language make it Turing powerful and what happens to its expressive power if priorities are introduced.

1 Introduction

Constraint is a ubiquitous concept: in every day life there are a lot of rules (physical, chemical, economical, and legal) that restrict, limit or regulate the way we operate and what decisions we take. In computer science constraints can be very useful not only to model the world but also to discover or verify if instances satisfy a model. For these reasons the notion of constraints gave birth to a new field called Constraint Programming that has attracted wide attention since it provides a concise and elegant way to describe problems and also efficient tools to compute solutions. Concurrency is a universal concept too. In every second of our life there are thousands of events or tasks occurring simultaneously and interacting with each other. With the evolution of the networks a lot of connected computers are available and nowadays more and more people think that we are inevitably going towards a world full of interconnected devices. This network of devices is a concurrent system and has peculiarities and characteristics that an environment constituted by only one processing unit does not have. On one hand a concurrent system is usually hard to use since, in such a system, problems like deadlocks, resources conflicts, security emerge. On the other hand a concurrent system can be the only mean to solve problems requiring huge amounts of resources or modeling complex scenarios in a simple and clear way.

The idea of approaching hard combinatorial optimization problems through a combination of search and constraint solving appeared first in logic programming. Despite the continued support of logic programming for constraint programmers, research efforts were initiated to import constraint technologies into other paradigms. Nowadays constraints can for instance be easily used in imperative languages. However constraints are equally well suited for modeling and supporting concurrency. In particular, concurrent computation can be seen for instance as agents that communicate and coordinate through a shared constraint store [21]. Importing constraint in existing languages raises some concerns: how easy is to use the new language, how expressive, and how extensible? Each concern is intrinsically linked to the host language and has a direct impact on potential end-users. It is desirable to obtain a declarative reading of a high-level model statement that exploits the facilities of the host language. Extensibility is also crucial since it is important to support the addition of user-defined constraints and user-defined search procedures.

Among all the concurrent languages having constraints as primitive building blocks [9, 23, 26–28, 30, 33] we focus our attention to the language called Constraint Handling Rule (CHR) [15, 16], a committed-choice declarative language. In the last few years, several papers have been devoted to investigate the expressivity of CHR that, as a language, is Turing powerful. When looking for decidable properties it is natural to consider restrictions of CHR which allow for instance the use of only the equality constraint = (interpreted in the usual way as equality on the Herbrand universe) and which, similarly to Datalog, is defined over a signature which contains no function symbols of arity > 0. In this work we first present, as an example, a decidability result obtained for a fragment of CHR that does not allow the introduction of new variables. Then we give an overview of the state of the art concerning the study of the decidability result of some fragments of CHR.

In the second part of the work we consider an extension of CHR dubbed CHR^{ω_p} [22] that was proposed for supporting an high-level, explicit form of execution control which is more flexible and declarative than the one offered by the usual ω_r semantics [12] of CHR. This extension is obtained by introducing explicitly in the syntax of the language rule annotations which allow to specify the priority of each rule to execute. 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. In this work we show several expressivity results relating CHR and CHR^{ω_p} by using the notion of acceptable encoding, i.e. a notion of expressivity coming from the field of concurrency theory. In fact, in this field the issue of the expressive power of a language has received a 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 can not 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 our algorithms. We show in particular an example of acceptable encoding that proves that dynamic priorities does not improve the expressive power of static priorities while instead priorities do improve the expressivity of CHR.

Structure of the paper. In Section 2 we introduce the CHR language and the notion of acceptable encoding used to discriminate between two Turing powerful languages. In Section 3 we study the decidability fragments of CHR while in Section 4 we investigate the expressive power of CHR with priorities. We conclude in Section 5.

Note. This work is an extract of the results presented in the PhD thesis of the author. For more details we defer the readers attention to [24]. I would like to thank my coauthors and my PhD advisor, Prof. Maurizio Gabbrielli, for their help and guidance during these last years.

2 Constraint Handling Rules

Constraint Handling Rule (CHR) [4, 14–16] is a committed-choice declarative language which has been originally designed for writing constraint solvers and which is nowadays a general purpose language. In this section, after introducing the used notation, we give an overview of the CHR syntax and semantics. We then define what is an acceptable encoding between two CHR languages.

2.1 Notation

Following [12, 16], we 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.

Definition 1 (Built-in constraint)**.** *A* built-in constraint $p(t_1, \ldots, t_n)$ *is an atomic predicate where* p *is a predicate symbol from* Π_b *and* $t_1, \ldots, t_n$ *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 $p(t_1, \ldots, t_n)$ *is an atomic predicate where* p *is a predicate symbol from* Π_u *and* $t_1, \ldots, t_n$ *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 will call these generally constraints). The capital versions of these notations will be used to denote multisets of constraints. We also denote by $false$ any inconsistent conjunction of constraints and with *true* the empty multiset of built-in constraints.

We will use "," rather than $\wedge$ to denote conjunction and we will 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 will 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 $\bar{t} = t_1, \ldots t_m$ and $\bar{t'} = t'_1, \ldots t'_m$ are sequences of terms then the notation $p(\bar{t}) = p'(\bar{t'})$ represents the set of equalities $t_1 = t'_1, \ldots, t_m = t'_m$ if $p = p'$, and it is undefined otherwise. This notation is extended in the expected way to multiset of constraints. Moreover we use $+\!\!+$ to denote sequence concatenation and $\uplus$ for multi-set 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 constraints 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 $\texttt{chr}(k\#i) = k$ and $\texttt{id}(k\#i) = i$, possibly extended to sets and sequences of identified CHR constraints in the obvious way.

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:

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

reflexivity	$\texttt{leq}(X,Y) \iff X = Y \mid true$
antisymmetry	$\texttt{leq}(X,Y), \texttt{leq}(Y,X) \iff X = Y$
transitivity	$\texttt{leq}(X,Y), \texttt{leq}(Y,Z) \Rightarrow \texttt{leq}(X,Z)$

Figure 1: A program for defining $\leq$ in CHR

where r is a unique identifier of a rule, H^k and H^h (the heads) are multi-sets of CHR constraints, D (the guard) is a possibly empty multi-set of built-in constraints and B is a possibly empty multi-set 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 multi-set of (both user-defined and built-in) constraints. An example of a CHR program is shown in Figure 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 [12] 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 multi-set 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 constraints), T is the propagation history (a set of sequence 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 propagation rule. The transitions 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 allows to update the constraint store by taking into account a built-in constraint contained in the goal. The **Introduce** transition is used to move a user-defined constraint from the goal

Solve	$\langle (c, G), S, C, T\rangle_n \xrightarrow{\omega_t}_P \langle G, S, c \wedge C, T\rangle_n$ where c is a built-in constraint
Introduce	$\langle (k, G), S, C, T\rangle_n \xrightarrow{\omega_t}_P \langle G, \{c\#n\} \cup S, C, T\rangle_{n+1}$ where k is a CHR constraint
Apply	$\langle G, H_1 \cup H_2 \cup S, C, T\rangle_n \xrightarrow{\omega_t}_P \langle (B, G), H_1 \cup S, \theta \wedge D \wedge C, T \cup \{t\}\rangle_n$ where P contains a (renamed apart) rule $$r \ @ H'_1 \backslash H'_2 \iff D \mid B$$ 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 $t = \mathtt{id}(H_1) +\!\!+ \mathtt{id}(H_2) +\!\!+ [r] \notin T$

Table 1: Transitions of ω_t

to the CHR constraint store, where it can be handled by applying CHR rules. The **Apply** transition 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 assumes that all variables appearing in a program clause are fresh ones. The **Apply** transition 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 $\mathtt{id}(H_1) +\!\!+ \mathtt{id}(H_2) +\!\!+ [r] \notin T$ avoids repeated application of the same 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 [16]] 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:*

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

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 [16] (a goal which has a data sufficient answer is called a data-sufficient goal in [16]).

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:*

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

Both previous notions of observables characterize an input/output behaviour, 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 Abstract operational semantics

The first CHR operational semantics defined in [16] differs from the traditional semantics ω_t. Indeed this original, so called, abstract semantics denoted by ω_a, allows the firing of a propagation rule an infinite number of times. For this reason ω_a can be seen as the abstraction of the traditional semantics where the propagation history is not considered. In Table 2 we have reported the transaction of the ω_a semantics following the structure of the theoretical semantics using configurations without a propagation history set.

Given a program P, the transition relation $\overset{\omega_a}{\rightarrow}_P \subseteq Conf \times Conf$ is the least relation satisfying the rules in Table 2.

Initial and final configurations can be defined analogously to those of ω_t semantics. In the same way we can define the observables: qualified and data sufficient answers.

2.5 CHR with priorities

De Koninck et al. [22] extended CHR with user-defined priorities. This new language, denoted by CHR^{ω_p}, provides an high level alternative for controlling program execution, that is more appropriate to 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

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

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

Table 2: Transitions of ω_a

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

Figure 2: A program for computing the shortest path in CHR^{ω_p}

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^{ω_p}, defined by [22], is an adaptation of the traditional semantics to deal with rule priorities. Formally this semantics, denoted by ω_p, is a state transition system $T = (Conf, \xrightarrow{\omega_p}_P)$ where P is a CHR^{ω_p} program while 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 $\xrightarrow{\omega_p}_P \subseteq Conf \times Conf$ is the least relation satisfying the rules in Table 3. The **Solve** and **Introduce** transitions are equal to those defined for the traditional semantics. The **Apply** transition instead is modified in order to take into account priorities. In fact, a further condition is added imposing 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 will use the terminology "higher" or "lower" priority rather than considering the values).

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	$\langle (k, G), S, C, T\rangle_n \overset{\omega_p}{\rightarrow}_P \langle G, \{c\#n\} \cup S, C, T\rangle_{n+1}$ 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 $$p :: r \ @ H_1' \backslash H_2' \iff D \mid B$$ 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

Table 3: Transitions of ω_p

An example of a CHR^{ω_p} program (from [22]) is shown in Figure 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, 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^{ω_p} can be defined analogously to those of the standard language:

Definition 5. *[Qualified answers] Let P be a CHR^{ω_p} 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:*

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

Definition 6. *[Data sufficient answers] Let P be a CHR^{ω_p} 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:

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

2.6 Language encoding

In order to compare the expressive power of two languages we use the notion of language encoding, first formalized in [10, 29, 34].[1] 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.

In this work we impose two requirements on the translation. First 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.

We formally define a *program encoding* as any function $\mathcal{PROG} : \mathcal{P}_{\mathcal{L}} \rightarrow \mathcal{P}_{\mathcal{L}'}$ which translates a $\mathcal{L}$ 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 $\mathcal{CT}$.

In order to define when an encoding is acceptable, 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 behaviour and the effects of the constraint theory).

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

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

Definition 7 (Acceptable encoding)**.** *Suppose that 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, $\mathcal{INP} : C \to C$ is the goal encoding, and $\mathcal{OUT} : C \to C$ is the output decoding which satisfy the following conditions:*

1. *the goal encoding function is compositional, that is, for any goal $(A, B) \in C$, $\mathcal{INP}(A, B) = \mathcal{INP}(A), \mathcal{INP}(B)$ 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 $(A, B) \in C$, $\mathcal{OUT}(A, B) = \mathcal{OUT}(A), \mathcal{OUT}(B)$ 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 C, that is, for all $P \in \mathcal{P}_{\mathcal{L}}$ and $G \in C$, $\mathcal{QA}_P(G) = \mathcal{OUT}(\mathcal{QA}_{\mathcal{PROG}(P)}(\mathcal{INP}(G)))$ 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 C, that is, for all $P \in \mathcal{P}_{\mathcal{L}}$ and $G \in C$, $\mathcal{SA}_P(G)$ 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 of the translation of A and B does not seem reasonable, as conjunction is the only form for goal composition available in CHR.

3 Non Turing powerful fragments of CHR

The computational power of CHR depends on several aspects, including the number of atoms allowed in the heads, the underlying signature Σ on which programs are defined, and the constraint theory $\mathcal{CT}$, defining the built-ins.

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

In this section, as an example, we give the proof that the fragment of CHR that does not introduce new variables is non Turing powerful. Then we provide an overview of the decidability results of CHR fragments.

The language under consideration in this section is the CHR defined over a signature which contains no function symbol of arity > 0 and interpreted using the ω_a semantics. We denote this language as $CHR^{\omega_a}(C)$. We also use the notation $CHR^{\omega_a}(P)$ to denote the language where all constraints have arity zero (i.e. $\Sigma = \emptyset$). Finally $CHR^{\omega_a}(F)$ indicates the CHR language which allows functor symbols and the = built-in.[3]

3.1 Range-restricted $CHR^{\omega_a}(C)$

We consider the (multi-headed) range-restricted $CHR^{\omega_a}(C)$ language. We call a CHR rule range-restricted if all the variables which appear in the body and in the guard appear also in the head of a rule. More formally, if $Var(X)$ denotes the variables used in X, the rule $r\ @H^k \backslash H^h \iff D \mid B$ is range-restricted if $Var(B) \cup Var(D) \subseteq Var(H^k, H^h)$ holds. A CHR language is called range-restricted if it allows range-restricted rules only.

We prove that in range-restricted $CHR^{\omega_a}(C)$ the existence of an infinite computation is a decidable property. This shows that this language is less expressive than Turing Machines and than $CHR^{\omega_a}(C)$. Our result is based on the theory of well-structured transition systems (WSTS) and we refer to [1, 13] for this theory. Here we only provide the basic definitions on WSTS, taken from [13].

Recall that a *quasi-order* (or, equivalently, preorder) is a reflexive and transitive relation. A *well-quasi-order* (wqo) is defined as a quasi-order $\leq$ over a set X such that, for any infinite sequence $x_0, x_1, x_2, \ldots$ in X, there exist indexes $i < j$ such that $x_i \leq x_j$. Thus well-quasi-orders exclude the possibility of having infinite strictly decreasing sequences.

A *transition system* is defined as usual, namely it is a structure $TS = (S, \rightarrow)$, where S is a set of *states* and $\rightarrow \subseteq S \times S$ is a set of *transitions*. We define $Succ(s)$ as the set $\{s' \in S \mid s \rightarrow s'\}$ of immediate successors of s. We say that TS is *finitely branching* if, for each $s \in S$, $Succ(s)$ is finite. Hence we have the key definition.

Definition 8 (Well-structured transition system with strong compatibility)**.** *A well-structured transition system with strong compatibility is a transition system $TS = (S, \rightarrow)$, equipped with a quasi-order $\leq$ on S, such that the two following conditions hold:*

1. *$\leq$ is a well-quasi-order;*

[3]Note that this last language is the signature used in most of the current CHR implementation. Indeed the host language of the majority of CHR implementations is Prolog and therefore the usual signature supports arbitrary Herbrand terms and unification.

2. $\leq$ *is strongly (upward) compatible with* $\rightarrow$*, that is, for all* $s_1 \leq t_1$ *and all transitions* $s_1 \rightarrow s_2$*, there exists a state* t_2 *such that* $t_1 \rightarrow t_2$ *and* $s_2 \leq t_2$ *holds.*

The next theorem is a special case of a result in [13] and will be used to obtain our decidability result.

Theorem 1. *Let* $TS = (S, \rightarrow, \leq)$ *be a finitely branching, well-structured transition system with strong compatibility, decidable* $\leq$ *and computable* $Succ(s)$ *for* $s \in S$*. Then the existence of an infinite computation starting from a state* $s \in S$ *is decidable.*

Consider a given goal G and a (CHR) program P and consider the transition system $T = (Conf, \overset{\omega_a}{\rightarrow}_P)$ defined in Section 2.4. Obviously the number of constants and variables appearing in G or in P is finite. Moreover, observe that since we consider range-restricted programs, the application of the transitions $\overset{\omega_a}{\rightarrow}_P$ does not introduce new variables in the computations. In fact, even though rules are renamed (in order to avoid clash of variables), the definition of the Apply rule (in particular the definition of θ) implies that in a transition $s_1 \overset{\omega_a}{\rightarrow}_P s_2$ we have that $Var(s_2) \subseteq Var(s_1)$ holds. Hence an obvious inductive argument implies that no new variables arise in computations. For this reason, given a goal G and a program P, we can assume that the set $Conf$ of all the configurations uses only a finite number of constants and variables. In the following we implicitly make this assumption. We define a quasi-order on configurations as follows.

Definition 9. *Given two configurations* $s_1 = \langle G_1, S_1, B_1 \rangle_i$ *and* $s_2 = \langle G_2, S_2, B_2 \rangle_j$ *we say that* $s_1 \leq s_2$ *if*

- *for every constraint* $c \in G_1$ $|\{c \in G_1\}| \leq |\{c \in G_2\}|$
- *for every constraint* $c \in \{d \,.\, d\#i \in S_1\}$ $|\{i \,.\, c\#i \in S_1\}| \leq |\{i \,.\, c\#i \in S_2\}|$
- B_1 *is logically equivalent to* B_2

It is possible to prove that $\leq$ is a well-quasi-order on $Conf$ and that given a $CHR^{\omega_a}(C)$ program P, $(Conf, \overset{\omega_a}{\rightarrow}_P, \leq)$ is a well-structured transition system with strong compatibility. For more details please see [20].

Exploiting the WSTS we have the desired result.

Theorem 2. *Given a range-restricted* $CHR^{\omega_a}(C)$ *program* P *and a goal* G*, the existence of an infinite computation for* G *in* P *is decidable.*

Proof. First observe that, due to our assumption on range-restricted programs, $T = (Conf, \overset{\omega_a}{\rightarrow}_P)$ is finitely branching. In fact, as previously mentioned, the use of rule Apply can not introduce new variables (and hence new different states). The thesis follows immediately from the strong compatibility of $(Conf, \overset{\omega_a}{\rightarrow}_P, \leq)$ and Theorem 1. □

The previous Theorem implies that range-restricted $CHR^{\omega_a}(C)$ is strictly less expressive than Turing Machines, in the sense that there can not exist a termination preserving encoding of Turing Machines into range-restricted $CHR^{\omega_a}(C)$. To be more precise, we consider an encoding of a Turing Machine into a CHR language as a function f which, given a machine Z and an initial instantaneous description D for Z, produces a CHR program and a goal. This is denoted by $(P, G) = f(Z, D)$. Hence we have the following.

Definition 10 (Termination preserving encoding)**.** *An encoding f of Turing Machines into a CHR language is termination preserving*[4] *if the following holds: the machine Z starting with D terminates iff the goal G in the CHR program P has only terminating computations, where $(P, G) = f(Z, D)$. The encoding is weak termination preserving if: the machine Z starting with D terminates iff the goal G in the CHR program P has at least one terminating computation.*

Since termination is undecidable for Turing Machines, we have the following immediate corollary of Theorem 2.

Corollary 1. *There exists no termination preserving encoding of Turing Machines into range-restricted $CHR^{\omega_a}(C)$.*

Note that the previous result does not exclude the existence of weak encodings. For example, in [8] it is shown that the existence of an infinite computation is decidable in CCS!, a variant of CCS, yet it is possible to provide a weak termination preserving encoding of Turing Machines in CCS! (essentially by adding spurious non-terminating computations). We conjecture that such an encoding is not possible for $CHR^{\omega_a}(C)$. Note also that previous results imply that range-restricted $CHR^{\omega_a}(C)$ is strictly less expressive than $CHR^{\omega_a}(C)$: in fact there exists a termination preserving encoding of Turing Machines into $CHR^{\omega_a}(C)$ [11, 31].

3.2 Overview

We proved that range-restricted $CHR^{\omega_a}(C)$ is strictly less expressive than Turing Machines (and therefore than $CHR^{\omega_a}(C)$). In [20] a similar result is proven for

[4]For many authors the existence of a termination preserving encoding into a non-deterministic language L is equivalent to the Turing completeness of L, however there is no general agreement on this, since for others a weak termination preserving encoding suffices.

$CHR_1^{\omega_a}(C)$, the language defined over a signature which does not allow function symbols but only one constraint in the head. These results are not immediate. Indeed, $CHR^{\omega_a}(C)$, without further restrictions and with any of the two semantics, is a Turing complete language [11,31]. It remains quite expressive also with these restrictions: for example, $CHR_1^{\omega_a}(C)$ allows an infinite number of different states, hence, for example, it can not be translated to Petri Nets.

Several papers have considered the expressive power of CHR in the last few years. In particular, [31] showed that a further restriction of $CHR_1^{\omega_a}(C)$, which does not allow built-ins in the body of rules (and which therefore does not allow unification of terms) is not Turing complete. This result is obtained by translating $CHR_1^{\omega_a}(C)$ programs (without unification) into propositional CHR and using the encoding of propositional CHR intro Petri Nets provided in [2]. The translation to propositional CHR is not possible for the language (with unification) $CHR_1^{\omega_a}(C)$. [2] also provides a translation of range-restricted $CHR^{\omega_a}(C)$ to Petri nets. However in this translation, differently from our case, it is also assumed that no unification built-in can be used in the rules, and only ground goals are considered. Related to this work is also [11], where it is shown that $CHR^{\omega_a}(F)$ is Turing complete and that restricting to single-headed rules decreases the computational power of CHR. However, these results are based on the theory of language embedding, developed in the field of concurrency theory to compare Turing complete languages, hence they do not establish any decidability result. Another related study is [32], 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. Earlier works by Frühwirth [17, 18] 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.

A summary of the existing results concerning the computational power of several dialects of CHR is shown in Table 4. In this table, "no" and "yes" refer to the existence of a termination preserving encoding of Turing Machines into the considered language.

4 Expressive power of priorities in CHR

In this section we first show 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 with priorities into the fragment of CHR that uses static priorities only. We then prove that (static) priorities augment the expressive power of CHR in the sense that there exists no acceptable encoding of CHR^{ω_p} into

Signature	*Operational semantics*	$k = 1$	$k > 1$
P (propositional)	ω_a	No	No
range-restricted C (constants)	ω_a	No	No
C (constants), without =	ω_a and ω_t	No	Yes
C (constants)	ω_a and ω_t	No	Yes
F (functors)	ω_a and ω_t	Yes	Yes

Table 4: Summary of termination preserving encoding of Turing Machines

CHR (with the ω_t semantics). To conclude we give an overview of the state of the art related to the expressive power equivalence between different CHR languages.

We consider the following languages and semantics:

- CHR^{ω_t}: this is standard CHR, where the theoretical semantics is used,
- CHR^{ω_p}: this is CHR with priorities, where both dynamic and static priorities can be used, the semantics is ω_p defined in Section 2.5;
- *static* CHR^{ω_p}: this is CHR with static priorities only, with the ω_p semantics;
- *static* $CHR_2^{\omega_p}$: 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.

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.

4.1 Encoding CHR^{ω_p} into *static* CHR^{ω_p}

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

We assume that P is a CHR^{ω_p} program composed by m rules and we also assume that the i-th rule (with $i \in \{1, \ldots, m\}$) has the form:

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

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

First, we require that the goal encoding is a non surjective function. The reason for this requirement is that the program encoding 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

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

The program encoding $\mathcal{T}(P)$ from CHR^{ω_p} into *static* CHR^{ω_p} is instead defined as the function that, given a program P, produces the following program:

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

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

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

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

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

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

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

If $rule_i$ is not a propagation rule then $\texttt{EVALUATE\_PRIORITIES}(i)$ are the following rules

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

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

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

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

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

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

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

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

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 $Update(C, V)$ function is defined instead as follows

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

$$Update(c(\bar{t}), V) = c(\bar{t})$$
if $c(\bar{t})$ 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 4.1. *L*et us consider as P the shortest path program depicted in Figure 2. The correspondent $\mathcal{T}(P)$ is the following program:

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

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

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

$$3 :: end \backslash instance_1(\bar{Z}) \Leftrightarrow true$$
$$3 :: end \backslash instance_2(\bar{Z}) \Leftrightarrow true$$
$$3 :: end \backslash instance_3(\bar{Z}) \Leftrightarrow true$$

$$4 :: end \Leftrightarrow true$$

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

$$6 :: new_{adist}(V_1, X_1, X_2), new_{adist}(V_2, Y_1, Y_2) \backslash highest\_priority(inf) \Leftrightarrow$$
$$X_2 \leq Y_2 | highest\_priority(1)$$
$$6 :: new_{adist}(V_1, X_1, X_2), new_{adist}(V_2, Y_1, Y_2) \backslash highest\_priority(P) \Leftrightarrow$$
$$X_2 \leq Y_2, 1 < P | highest\_priority(1)$$

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

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

$$8 :: new_{asource}(V, X) \backslash instance_1(\bar{V}), highest\_priority(P), id(V') \Leftrightarrow$$
$$1 = P | new_{adist}(V', X, 0), id(V'+1), highest\_priority(inf)$$
$$8 :: new_{adist}(V_1, X, X_1) \backslash new_{adist}(V_2, X, X_2), highest\_priority(P), id(V') \Leftrightarrow$$
$$X_1 \leq X_2, 1 = P | id(V'), highest\_priority(inf)$$
$$8 :: new_{adist}(V_1, X, X_1), new_{aedge}(V_2, X, X_2, X_3) \backslash instance_3(V_1, V_2), highest\_priority(P),$$
$$id(V') \Leftrightarrow X_1 + 2 = P | new_{adist}(X_3, X_1 + X_2), id(V'+1), highest\_priority(inf)$$

□

We now provide some explanations for the above encoding. Intuitively the result of the encoding can be divided in three phases:

1. **Init.** 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 $ak(\bar{t})$ distinct variables) 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

  $$\texttt{EVALUATE\_PRIORITIES}(i)$$

  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

  $$\texttt{EVALUATE\_PRIORITIES}(i)$$

  should consider the possibility that there is an instance of $rule_i$ that can not 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 $rule_i$ can not 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.

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 $\mathcal{T}(P)$ 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 $\mathcal{T}(P)$ 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 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 result proven in [19] shows that the qualified answers are preserved by our encoding.

Theorem 3. *The triple ($\mathcal{T}()$, $\mathcal{INP}()$, $\mathcal{OUT}()$) provides an acceptable encoding between CHR^{ω_p} and static CHR^{ω_p}.*

4.2 No encoding of *static* CHR^{ω_p} into CHR^{ω_t}

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^{ω_p} 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 of 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, K) in P will terminate 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.*

Lemma 1 is not true anymore if we consider CHR^{ω_p} 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 4. *There exists no acceptable encoding for data sufficient answers from CHR^{ω_p} into CHR^{ω_t}. class $\mathcal{G}$.*

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

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

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

and assume that $(\gamma(), \mathcal{INP}(), \mathcal{OUT}())$ is an acceptable encoding for data sufficient answers from CHR^{ω_p} into CHR^{ω_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, $\mathcal{QA}_{\gamma(P)}(\mathcal{INP}(a(X)))$ 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^{ω_p} into CHR^{ω_t}, thus concluding the proof. □

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^{ω_p} into CHR^{ω_t}.*

4.3 Overview

Some immediate acceptable encodings derive directly from the language definitions. Indeed, 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. We first observe that *static* $CHR_2^{\omega_p}$ is a sublanguage of *static* CHR^{ω_p} that, in its turn, is a sublanguage of CHR^{ω_p}. Therefore we have the following.

Fact 1. *There exists acceptable encodings from static $CHR_2^{\omega_p}$ to static CHR^{ω_p}, and from static CHR^{ω_p} to CHR^{ω_p}.*

As far as CHR^{ω_t} is concerned, at a first glance it could be considered as a sublanguage of *static* CHR^{ω_p} 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^{ω_t} program P can be reached by the *static* CHR^{ω_p} 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^{ω_t} to static CHR^{ω_p}.*

In [19] it is 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. Moreover, as proven in Section 4.1, dynamic priorities do not increase the expressive power w.r.t. static ones.

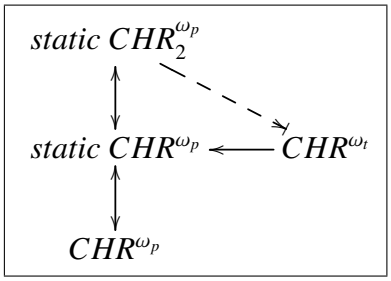

Figure 3: Graphical summary:
$\dashrightarrow$: absence of an acceptable encoding
$\rightarrow$: presence of an acceptable encoding

On the other hand we have proven 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. For a graphical overview of the expressive power of CHR with and without priority we refer to Figure 3.

Our notion of acceptable encoding has been recently used in [3] to justify a source-to-source transformation. When instead we move to the more general field of concurrent languages one can find several works related to the present one. In particular, concerning priorities, [35] shows 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 can not be encoded into π-calculus nor in the broadcast based b-π calculus. This result is related to our separation result for CHR^{ω_p} 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 [25] 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 in [36].

5 Conclusions

We considered Constraint Handling Rules (CHR), a well known concurrent language that supports constraints as primitive constructs. We studied its expressive power focusing first on some of its fragments and then considering what happens when priorities are added.

There are still plenty of open question to address. For instance one may won-

der if the expressive power of CHR with priorities is equal to the CHR with the refined semantics [12]. Another question to answer could be if range-restricted $CHR^{\omega_a}(C)$ is more expressive that $CHR_1^{\omega_a}(C)$, since the decidability result for the second language is stronger. Moreover there are hundreds of concurrent languages that can be enriched with constraint primitives to improve their expressive power. We are experiencing an increasingly attention towards this kind of tasks, e.g. [5–7], and we expect a continuation of this trend in the feature.

References

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *in Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, LICS'96*, pages 313–321, 1996.

[2] Hariolf Betz. Relating coloured Petri nets to Constraint Handling Rules. In K. Djelloul, G. J. Duck, and M. Sulzmann, editors, *4th Workshop on Constraint Handling Rules*, pages 33–47, Porto, Portugal, 2007.

[3] Hariolf Betz, Frank Raiser, and Thom W. Frühwirth. A complete and terminating execution model for Constraint Handling Rules. *TPLP*, 10(4-6):597–610, 2010.

[4] Stefano Bistarelli, Thom W. Frühwirth, and Michael Marte. Soft constraint propagation and solving in CHRs. In *SAC*, pages 1–5, 2002.

[5] Stefano Bistarelli and Francesco Santini. A Nonmonotonic Soft Concurrent Constraint Language for SLA Negotiation. *Electr. Notes Theor. Comput. Sci.*, 236:147–162, 2009.

[6] Maria Grazia Buscemi and Ugo Montanari. Open Bisimulation for the Concurrent Constraint Pi-Calculus. In Sophia Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2008.

[7] Maria Grazia Buscemi and Ugo Montanari. A survey of constraint-based programming paradigms. *Computer Science Review*, 2(3):137–141, 2008.

[8] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Comparing Recursion, Replication, and Iteration in Process Calculi. In *ICALP*, pages 307–319, 2004.

[9] Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In *TIME*, pages 227–233, 2001.

[10] Frank S. de Boer and Catuscia Palamidessi. Embedding as a tool for language comparison. *Inf. Comput.*, 108(1):128–157, 1994.

[11] Cinzia Di Giusto, Maurizio Gabbrielli, and Maria Chiara Meo. Expressiveness of multiple heads in CHR. In *SOFSEM*, pages 205–216, 2009.

[12] Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *ICLP*, pages 90–104, 2004.

[13] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[14] Thom Frühwirth. Temporal reasoning with Constraint Handling Rules. Technical Report ECRC-94-5, European Computer-Industry Research Centre, Munchen, Germany, 1994.

[15] Thom Frühwirth. *Constraint Handling Rules*. August 2009.

[16] Thom W. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Log. Program.*, 37(1-3):95–138, 1998.

[17] Thom W. Frühwirth. As Time Goes by: Automatic Complexity Analysis of Simplification Rules. In *KR*, pages 547–557, 2002.

[18] Thom W. Frühwirth and Slim Abdennadher. The Munich Rent Advisor: A Success for Logic Programming on the Internet. *TPLP*, 1(3):303–319, 2001.

[19] Maurizio Gabbrielli, Jacopo Mauro, and Maria Chiara Meo. The expressive power of chr with priorities. *Inf. Comput.*, 228:62–82, 2013.

[20] Maurizio Gabbrielli, Jacopo Mauro, Maria Chiara Meo, and Jon Sneyers. Decidability properties for fragments of CHR. *TPLP*, 10(4-6):611–626, 2010.

[21] Maurizio Gabbrielli, Catuscia Palamidessi, and Frank D. Valencia. Concurrent and Reactive Constraint Programming. In *25 Years GULP*, pages 231–253, 2010.

[22] Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In *PPDP*, pages 25–36, 2007.

[23] Michael J. Maher. Logic semantics for a class of committed-choice programs. In *ICLP*, pages 858–876, 1987.

[24] Jacopo Mauro. *Constraints meet concurrency*. PhD thesis, University of Bologna, 2012.

[25] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[26] Catuscia Palamidessi and Frank D. Valencia. A Temporal Concurrent Constraint Programming Calculus. In *CP*, pages 302–316, 2001.

[27] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Default Timed Concurrent Constraint Programming. In *POPL*, pages 272–285, 1995.

[28] Vijay A. Saraswat and Martin C. Rinard. Concurrent Constraint Programming. In *POPL*, pages 232–245, 1990.

[29] Ehud Y. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.

[30] Gert Smolka. The Oz Programming Model. In *Computer Science Today*, pages 324–343. 1995.

[31] Jon Sneyers. Turing-complete subclasses of CHR. In *ICLP*, pages 759–763, 2008.

[32] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. *ACM Trans. Program. Lang. Syst.*, 31(2), 2009.

[33] Kazunori Ueda. Guarded Horn Clauses. In *LP*, pages 168–179, 1985.

[34] Frits W. Vaandrager. Expressive Results for Process Algebras. In *REX Workshop*, pages 609–638, 1992.

[35] Cristian Versari, Nadia Busi, and Roberto Gorrieri. On the Expressive Power of Global and Local Priority in Process Calculi. In *CONCUR*, pages 241–255, 2007.

[36] Maria Grazia Vigliotti, Iain Phillips, and Catuscia Palamidessi. Tutorial on separation results in process calculi via leader election problems. *Theor. Comput. Sci.*, 388(1-3):267–289, 2007.