

Generalized CHR Machines

Jon Sneyers ^{*1} and Thom Frühwirth²

¹ K.U. Leuven, Belgium

`jon.sneyers@cs.kuleuven.be`

² University of Ulm, Germany

`thom.fruehwirth@uni-ulm.de`

Abstract. Constraint Handling Rules (CHR) is a high-level rule-based programming language. In [11], a model of computation based on the operational semantics of CHR is introduced, called the CHR machine. The CHR machine was used to prove a complexity-wise completeness result for the CHR language and its implementations. In this paper, we investigate three generalizations of CHR machines: CHR machines with an instantiated operational semantics, non-deterministic CHR machines, and self-modifying CHR machines.

1 Introduction

Constraint Handling Rules is a high-level language extension based on multi-headed committed-choice rules [5, 12, 6]. The abstract operational semantics ω_t of CHR is very non-deterministic. Since rule applications are committed-choice — CHR has no built-in search mechanisms like backtracking — confluence of a CHR program is a crucial property.

In earlier work [11] we have shown a complexity-wise completeness result for CHR: everything (every RAM-machine program) can be implemented in CHR (in a confluent way), and there are CHR systems which execute the resulting CHR program with the right time and space complexity. Recently, Di Giusto et al. have shown that even single-headed CHR rules suffice to implement a Turing-equivalent Minsky machine [9], although they argue that in terms of expressive power, multiple heads are still needed.

In the approach of [11], a theoretical “CHR machine” is defined, which performs one ω_t transition in every step. In this paper we propose several more general definitions for CHR machines. We first recapture some of the definitions of [11]. This leads to the definition of the original kind of CHR machines, which we now call *deterministic abstract CHR machines* (Section 2). Section 3 introduces the concept of *strategy classes*, which formalizes instantiations of the ω_t operational semantics. We also define and discuss a more general notion of confluence. Then, in Section 4, we consider CHR machines with an instantiated

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). This research was performed in large part while Jon Sneyers was visiting the University of Ulm.

operational semantics (for example the refined operational semantics [3]). Finally, in Section 5, we define non-deterministic CHR machines, and in Section 6, we define CHR machines with a stored program, also known as *self-modifying* CHR machines.

2 Deterministic Abstract CHR Machines

We assume the reader to be familiar with CHR. We will use the same notation as in [11]. That is, we denote the host language with \mathcal{H} , the built-in constraint theory with $\mathcal{D}_{\mathcal{H}}$, the set of queries for a program \mathcal{P} with $\mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$, the abstract (or theoretical) operational semantics with ω_t , its execution states with $\sigma, \sigma_0, \sigma_1, \dots$ and the set of all execution states with Σ^{CHR} , the ω_t transition relation with $\mapsto_{\mathcal{P}}$ and its transitive closure with $\mapsto_{\mathcal{P}}^*$.

2.1 Derivations

Definition 1. Given an initial goal (or query) $\mathbb{G} \in \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$, the corresponding initial state is defined as $\text{initstate}(\mathbb{G}) = \langle \mathbb{G}, \emptyset, \text{true}, \emptyset \rangle_1$.

We denote the set of all initial states by $\Sigma^{\text{init}} \subset \Sigma^{\text{CHR}}$.

Definition 2. A final state $\sigma_f = \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ is an execution state for which no transition applies: $\neg \exists \sigma \in \Sigma^{\text{CHR}} : \sigma_f \mapsto_{\mathcal{P}} \sigma$. In a failure state, the underlying solver \mathcal{H} can prove $\mathcal{D}_{\mathcal{H}} \models \neg \exists \mathbb{B}$ — such states are always final. A successful final state is a final state that is not a failure state, i.e. $\mathcal{D}_{\mathcal{H}} \models \exists \mathbb{B}$. The set of final states is denoted by $\Sigma^{\text{final}} \subset \Sigma^{\text{CHR}}$.

Definition 3. Given a CHR program \mathcal{P} , a finite derivation d is a finite sequence $[\sigma_0, \sigma_1, \dots, \sigma_n]$ of states where $\sigma_0 \in \Sigma^{\text{init}}$, $\sigma_n \in \Sigma^{\text{final}}$, and $\sigma_i \mapsto_{\mathcal{P}} \sigma_{i+1}$ for $0 \leq i < n$. If σ_n is a failure state, we say d has failed, otherwise d is a successful derivation.

Definition 4. An infinite derivation d_{∞} is an infinite sequence $\sigma_0, \sigma_1, \dots$ of states where $\sigma_0 \in \Sigma^{\text{init}}$ and $\sigma_i \mapsto_{\mathcal{P}} \sigma_{i+1}$ for $i \in \mathbb{N}$.

We use $\#d$ to denote the length of a derivation: the length of a finite derivation is the number of transitions in the sequence; the length of an infinite derivation is ∞ . A set of (finite or infinite) derivations is denoted by Δ . The set of all derivations in Δ that start with $\text{initstate}(\mathbb{G})$ is denoted by $\Delta|_{\mathbb{G}}$. We use $\Delta_{\omega_t}^{\mathcal{H}}(\mathcal{P})$ to denote the set of all derivations (in the ω_t semantics) for a given CHR program \mathcal{P} and host language \mathcal{H} . We now define the relation $\rightsquigarrow_{\Delta} : \Sigma^{\text{init}} \rightarrow \Sigma^{\text{CHR}} \cup \{\infty\}$:

Definition 5. State σ_n is a Δ -output of σ_0 if $[\sigma_0, \dots, \sigma_n] \in \Delta$. We say σ_0 Δ -outputs σ_n and write $\sigma_0 \rightsquigarrow_{\Delta} \sigma_n$. If Δ contains an infinite derivation starting with σ_0 , we say σ_0 has a non-terminating derivation. We denote this as follows: $\sigma_0 \rightsquigarrow_{\Delta} \infty$.

Definition 6. The CHR program \mathcal{P} is Δ -deterministic for input $I \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ if the restriction of $\rightsquigarrow_{\Delta}$ to $\text{initstate}[I]$ is a function and $\forall i \in I, d \in \Delta|_i$: if d is a successful derivation, then $\forall d' \in \Delta|_i$: $\#d = \#d'$.

In other words, a program is Δ -deterministic if all derivations starting from a given input have the same result and all successful ones have the same length. Note that if a program \mathcal{P} is $\Delta_{\omega_t}^{\mathcal{H}}(\mathcal{P})$ -deterministic for all input $\mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$, it is also observable confluent [4]. The notion of observable confluence does not require derivations to have the same length.

2.2 Deterministic Abstract CHR Machines

We now define a class of CHR machines, which corresponds to the definition given in [11]. This class of CHR machines is somewhat restricted since it only allows $\Delta_{\omega_t}^{\mathcal{H}}$ -deterministic CHR programs. In Section 4 we will allow more general CHR machines.

Definition 7. A deterministic abstract CHR machine is a tuple $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{V}\mathcal{G})$. The host language \mathcal{H} defines a built-in constraint theory $\mathcal{D}_{\mathcal{H}}$, \mathcal{P} is a CHR program, and $\mathcal{V}\mathcal{G} \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ is a set of valid goals, such that \mathcal{P} is a $\Delta_{\omega_t}^{\mathcal{H}}$ -deterministic CHR program for input $\mathcal{V}\mathcal{G}$. The machine takes an input query $\mathbb{G} \in \mathcal{V}\mathcal{G}$ and executes a derivation $d \in \Delta_{\omega_t}^{\mathcal{H}}|_{\mathbb{G}}$.

Terminology. If the derivation d for \mathbb{G} is finite, we say the machine *terminates* with *output state* $\mathcal{M}(\mathbb{G}) = \langle \mathbb{G}', \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ which is the last state of d . We will use the following notation: $\mathcal{M}_c(\mathbb{G}) = \mathbb{S}$ is the set of *output constraints*, $\mathcal{M}_b(\mathbb{G}) = \mathbb{B}$ is the *output built-in store*, and $\text{deriv}_{\mathcal{M}}(\mathbb{G}) = d$ is a *derivation for* \mathbb{G} : if the derivations are successful, $\text{deriv}_{\mathcal{M}}(\mathbb{G})$ returns an arbitrary derivation; if the derivations fail, $\text{deriv}_{\mathcal{M}}(\mathbb{G})$ returns one of the derivations of maximal length. The machine *accepts* the input \mathbb{G} if d is a successful derivation and *rejects* \mathbb{G} if d is a failed derivation ($\mathcal{M}_b(\mathbb{G}) = \text{fail}$). If d is an infinite derivation, we say the machine *does not terminate*. A *CHR(X) machine* is a CHR machine for which the host language $\mathcal{H} = X$. We use Φ to denote *no host language*: the built-in constraint theory \mathcal{D}_{Φ} defines only the basic constraints *true* and *fail*, and syntactic equality and inequality (only to be used as an *ask*-constraint). This implies that the **Solve** transition can only be used once (to add *fail*). The only data types are logical variables (that will not be bound) and constants. A *CHR-only machine* is a $\text{CHR}(\Phi)$ machine.

Definition 8. A sufficiently strong *host language* \mathcal{H} is a host language whose built-in constraint theory $\mathcal{D}_{\mathcal{H}}$ defines at least *true*, *fail*, `==` and `\==`, the integer numbers and the arithmetic operations for addition, subtraction, multiplication and integer division.

Clearly, most host languages are sufficiently strong. Prolog for instance allows arithmetic using the built-in `is/2`. In CHR program listings where the host

```

r1 @ delta(Q,S,Q2,S2,left), adj(LC,C) \ state(Q), cell(C,S), head(C)
    <=> LC \== null | state(Q2), cell(C,S2), head(LC).
r2 @ delta(Q,S,Q2,S2,right), adj(C,RC) \ state(Q), cell(C,S), head(C)
    <=> RC \== null | state(Q2), cell(C,S2), head(RC).
r3 @ delta(Q,S,Q2,S2,left) \ adj(null,C), state(Q), cell(C,S), head(C)
    <=> cell(LC,b), adj(null,LC), adj(LC,C), state(Q2), cell(C,S2), head(LC).
r4 @ delta(Q,S,Q2,S2,right) \ adj(C,null), state(Q), cell(C,S), head(C)
    <=> cell(RC,b), adj(C,RC), adj(RC,null), state(Q2), cell(C,S2), head(RC).
fail @ nodelta(Q,S), rejecting(Q), state(Q), cell(C,S), head(C) <=> fail.

```

Fig. 1. The CHR program TMSIM, a Turing machine simulator.

language is assumed to be sufficiently strong, we will use a slightly abbreviated notation. For example, if $\mathbf{c}/1$ is a CHR constraint, we write expressions like “ $\mathbf{c}(N+1)$ ”: a host language independent notation that is equivalent to “ M is $N+1$, $\mathbf{c}(M)$ ” for CHR(Prolog), to “ $\mathbf{c}(\text{intUtil.add}(N,1))$ ” for CHR(Java), etc.

2.3 Computational Power of CHR Machines

We assume the reader to be familiar with Turing machines; we refer to [11] for a more detailed exposition.

A model of computation is called *Turing-complete* if it has the same computational power as Turing machines: every Turing Machine can be simulated in the model and every program of the model can be simulated on a Turing machine. Consider the CHR program TMSIM shown in Figure 1 and the corresponding CHR-only machine $\mathcal{M}_{\text{TMSIM}} = (\emptyset, \text{TMSIM}, \mathcal{G}_{\text{TMSIM}})$. The program simulates Turing machines. Intuitively, the meaning of the constraints of TMSIM is as follows:

- delta/5** encodes the transition function (the Turing machine program) in the obvious way: the first two arguments are inputs, the last three are outputs;
- nodelta/2** encodes the domain on which δ is undefined;
- rejecting/1** encodes the set of non-accepting final states;
- state/1** contains the current state;
- head/1** contains the identifier of the cell under the head;
- cell/2** represents a tape cell. The first argument is the unique identifier of the cell. The second argument is the symbol in the cell.
- adj/2** encodes the order of the tape cells. The constraint **adj**(A, B) should be read: “the right neighbor of the tape cell with identifier A is the tape cell with identifier B ”. The special cell identifier **null** is used to refer to a not yet instantiated cell. The rules r_3 and r_4 extend the tape as needed.

The set of valid goals $\mathcal{G}_{\text{TMSIM}}$ corresponds to the goals that represent a valid Turing machine and a correctly represented tape; it is defined formally in [11].

A simulation of the execution of a Turing machine M proceeds as follows. The tape input is encoded as **cell/2** constraints and **adj/2** constraints. The identifier of the cell to the left of the left-most input symbol is set to **null** and

similarly for the cell to the right of the right-most input symbol. The transition function δ of M is encoded in multiple **delta**/5 constraints. All these constraints are combined in the initial query together with the constraint **state**(q_0) where q_0 is the initial state of M and the constraint **head**(c_1) where c_1 is the identifier of the cell representing the left-most input symbol. Every rule application of the first four rules of TMSIM corresponds directly to a Turing machine transition.

If no more (Turing machine) transitions can be made, the last rule is applicable if the current state is non-accepting. In that case, the built-in constraint *fail* is added, which leads to a failure state. If the Turing machine ends in an accepting final state, the CHR program ends in a successful final state.

The program TMSIM can easily be rewritten to use only simplification rules. A Turing machine simulator can also be written using only propagation rules, or using only single-headed simplification rules, or with only guardless rules.

2.4 Time and Space Complexity

We define the time complexity of a CHR machine in an obvious way:

Definition 9. *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{V}\mathcal{G})$, the function $chrtime_{\mathcal{M}}$ returns the derivation length, given a valid goal:*

$$chrtime_{\mathcal{M}} : \mathcal{V}\mathcal{G} \rightarrow \mathbb{N} : \mathbb{G} \mapsto \max\{\#d \mid d \in \Delta_{\omega_t}^{\mathcal{H}}|_{\mathbb{G}}\}.$$

Definition 10. *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{V}\mathcal{G})$ and assuming that host language constraints of \mathcal{H} take constant time, the (worst-case) time complexity function $CHRTIME_{\mathcal{M}}$ is defined as follows:*

$$CHRTIME_{\mathcal{M}}(n) = \max\{chrtime_{\mathcal{M}}(\mathbb{G}) \mid \mathbb{G} \in \mathcal{V}\mathcal{G} \wedge inputsize(\mathbb{G}) = n\}$$

where *inputsize* is a function which returns the size of a goal.

Note that the definition of $chrtime_{\mathcal{M}}$ does not correspond to what is obtainable in real CHR implementations, because finding an applicable rule with k heads in a store of size n may take up to $\mathcal{O}(n^k)$ time.

Definition 11 (State size function).

$$SIZE : \Sigma^{\text{CHR}} \rightarrow \mathbb{N} : \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto size(\mathbb{G}) + size(\mathbb{S}) + size(\mathbb{B}) + size(\mathbb{T})$$

where for sets X , $size(X) = \sum_{x \in X} |x|$ and the size $|x|$ is the usual term size.

Definition 12. *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{V}\mathcal{G})$, the space function $chrspac_{\mathcal{M}}$ returns the worst-case execution state size, given an initial goal:*

$$chrspac_{\mathcal{M}}(\mathbb{G}) = \max\{SIZE(\sigma) \mid \sigma \in \Delta_{\omega_t}^{\mathcal{H}}|_{\mathbb{G}}\}.$$

Definition 13. *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{V}\mathcal{G})$, the (worst-case) space complexity function $CHRSPACE_{\mathcal{M}}$ is defined as follows:*

$$CHRSPACE_{\mathcal{M}}(n) = \max\{chrspac_{\mathcal{M}}(\mathbb{G}) \mid \mathbb{G} \in \mathcal{V}\mathcal{G} \wedge inputsize(\mathbb{G}) = n\}$$

Note that if the size of individual constraints is bounded, the size of the constraint store is asymptotically dominated by the number of **Introduce** steps, and the size of the built-in store is dominated by the number of **Solve** steps of the ω_t operational semantics.

2.5 Complexity of CHR Machines

In [11] we have demonstrated that there is a deterministic abstract CHR machine that can simulate RAM machines (and hence also Turing machines) without any overhead. This means that “everything can be done efficiently in CHR”.

Theorem 1. *For any sufficiently strong host language \mathcal{H} , a $\text{CHR}(\mathcal{H})$ machine \mathcal{M}_{RAM} exists which can simulate, in $\mathcal{O}(T + P + S)$ time and $\mathcal{O}(S + P)$ space, a T -time, S -space standard RAM machine with a program of P lines.*

We have also demonstrated that everything can be done efficiently in CHR *in practice*, in the sense that the CHR program can be executed in existing implemented CHR systems with the right time and space complexity:

Theorem 2. *For every (RAM machine) algorithm which uses at least as much time as it uses space, a CHR program exists which can be executed in the K.U.Leuven CHR system, with time and space complexity within a constant from the original complexities.*

3 Strategy Classes

The ω_t operational semantics is very nondeterministic, in the sense that for most programs, the number of possible derivations is very large. This is of course the reason why the confluence property is crucial for program correctness.

However, all CHR implementations somehow restrict the nondeterminism of the ω_t semantics. While still guaranteeing rule application until exhaustion, they usually impose some order in which rules are tried. In effect, they instantiate the ω_t semantics; the best-known such instantiation is the refined operational semantics ω_r [3].

In this section we examine instantiations of the ω_t semantics in a general formal framework. We hope that this will lead to more insight and intuition, and this framework may also be used to study the effects of differences in implementations.

3.1 Execution Strategies

Definition 14. *An execution strategy fixes the output for every initial state. Formally, ξ is an execution strategy for a program \mathcal{P} if $\xi \subseteq \Delta_{\omega_t}^{\mathcal{H}}(\mathcal{P})$ and \rightsquigarrow_{ξ} is a total function over Σ^{init} , i.e.*

$$\forall x, y, z \in \Sigma^{\text{CHR}} : x \rightsquigarrow_{\xi} y \wedge x \rightsquigarrow_{\xi} z \Rightarrow y = z \quad (1)$$

$$\forall \sigma \in \Sigma^{\text{init}} : \exists \sigma' \in \Sigma^{\text{CHR}} \cup \{\infty\} : \sigma \rightsquigarrow_{\xi} \sigma' \quad (2)$$

The set of all execution strategies for a program \mathcal{P} is denoted by $\Omega_t^{\mathcal{H}}(\mathcal{P})$. For a given program, the set of all derivations $\Delta_{\omega_t}^{\mathcal{H}}(\mathcal{P})$ is countably infinite. Since $\Omega_t^{\mathcal{H}}(\mathcal{P})$ is essentially the powerset of $\Delta_{\omega_t}^{\mathcal{H}}(\mathcal{P})$, it is uncountable. Since there are only countably infinitely many computer programs, clearly not every execution strategy can be implemented. For instance, with the right execution strategy, the following program solves the halting problem:

```

check_halts(TM) <=> true.
check_halts(TM) <=> fail.

```

Clearly we need a more realistic notion of execution strategies.

Definition 15. A computable execution strategy ξ is an execution strategy for which the following objects exist: a set of concrete states $\mathcal{C}\Sigma$, a computable abstraction function $\alpha : \mathcal{C}\Sigma \rightarrow \Sigma^{\text{CHR}}$, a computable initialization function $\beta : \Sigma^{\text{init}} \rightarrow \mathcal{C}\Sigma$ such that $\forall \sigma \in \Sigma^{\text{init}} \alpha(\beta(\sigma)) = \sigma$, and a computable partial concrete transition function $\mathcal{C}t : \mathcal{C}\Sigma \rightarrow \mathcal{C}\Sigma$, such that, $\forall x \in \Sigma^{\text{init}}$:

$$x \rightsquigarrow_{\xi} y \iff \exists n \in \mathbb{N} : \alpha(\mathcal{C}t^n(\beta(x))) = y \text{ and } \mathcal{C}t^{n+1}(\beta(x)) \text{ is undefined}$$

A function is computable if there is a Turing machine that computes it.

3.2 Strategy Classes

An execution strategy completely determines the result of any query. In general, although the result set may be smaller than that of all ω_t derivations, most CHR systems are still not completely deterministic. For instance, the refined operational semantics does not fix the order of constraint reactivation and partner constraint matching, which could lead to different results for the same query. However, a specific version of a CHR system, possibly with specific compiler optimizations turned on or off, should be completely deterministic, so it has only one execution strategy for every CHR program.

Definition 16. A strategy class $\Omega(\mathcal{P}) \subseteq \Omega_t^{\mathcal{H}}(\mathcal{P})$ is a set of execution strategies for \mathcal{P} .

As an example, there is a strategy class corresponding to the K.U.Leuven CHR system (which may contain more than one execution strategy depending on the version and the settings of the optimization flags), and it is a subset of the strategy class corresponding to the refined operational semantics. Many CHR implementations are (possibly different) instantiations of the refined semantics, in the sense that their strategy class is a subset of $\Omega_r^{\mathcal{H}}$.

Note that strategy classes are subsets of the power set of $\Delta_{\omega_t}^{\mathcal{H}}$. We will sometimes drop the argument (\mathcal{P}) to avoid heavy notation.

Definition 17. A computable strategy class is a strategy class which contains at least one computable execution strategy.

Clearly, the K.U.Leuven CHR strategy class is computable, which implies that the refined semantics is also a computable strategy class, and so is the abstract semantics.

The refined operational semantics ω_r . We define Σ_r^{CHR} to be the set of execution states of the refined semantics [3], and $\Omega_r^{\mathcal{H}}(\mathcal{P})$ as the strategy class corresponding to ω_r derivations of a program \mathcal{P} .

Not all execution strategies in $\Omega_r^{\mathcal{H}}(\mathcal{P})$ are computable. As an example, consider the following program:

```

check_halts(TM) <=> answer(true), answer(false), choose.
choose, answer(X) <=> do(X).
do(fail) <=> fail.

```

With an appropriate ω_r execution strategy, this program solves the halting problem. Since the second rule is applied with `choose/0` as the active constraint, the refined semantics does not fix the choice of partner constraint. The execution strategy that always picks the correct answer is obviously not computable.

To show that $\Omega_r^{\mathcal{H}}(\mathcal{P})$ is a computable strategy class, it suffices to identify one computable execution strategy in $\Omega_r^{\mathcal{H}}(\mathcal{P})$. Following [3], we can define an abstraction function α which maps Σ_r^{CHR} to Σ^{CHR} :

$$\alpha(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \text{no_id}(\mathbb{A}), \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$$

where $\text{no_id}(\mathbb{A}) = \{c \mid c \in \mathbb{A} \text{ is not of the form } c\#i \text{ or } c\#i : j\}$. The set of concrete states is Σ_r^{CHR} and the initialization function is the identity function.

Now we still have not given a computable execution strategy, which needs to have a transition relation that is a computable function, that is, every state has a unique next state, and the latter can be computed from the former. The transition relation of ω_r is not a function, so we consider a subset of it which is a function, for instance, the function which maps every execution state in $\sigma \in \Sigma_r^{\text{CHR}}$ to the lexicographically first element in $\{\sigma' \mid \sigma \mapsto^{\omega_r} \sigma'\}$. Since the set of all next ω_r states is computable, and the lexicographically first element of a set is computable, this function is computable.

The priority semantics ω_p . We define $\Omega_p^{\mathcal{H}}(\mathcal{P})$ as the strategy class corresponding to derivations in the priority semantics ω_p [1]. We denote an assignment of priorities to rules with \bar{p} , and we write $\Omega_p^{\mathcal{H}}(\mathcal{P}, \bar{p})$ for the subset of $\Omega_p^{\mathcal{H}}(\mathcal{P})$ which corresponds to ω_p derivations with the priority assignments \bar{p} .

Again, $\Omega_p^{\mathcal{H}}(\mathcal{P})$ contains non-computable execution strategies as well as computable ones.

3.3 Generalized Confluence

We now generalize the definition of confluence to arbitrary strategy classes:

Definition 18 (Ω -confluence). A CHR program \mathcal{P} is $\Omega(\mathcal{P})$ -confluent if, for every initial state $\langle \mathbb{G}, \emptyset, \text{true}, \emptyset \rangle_1 = \sigma \in \Sigma^{\text{init}}$ and execution strategies $\xi_1, \xi_2 \in \Omega(\mathcal{P})$, the following holds:

$$\left. \begin{array}{l} \sigma \rightsquigarrow_{\xi_1} \langle \mathbb{G}_1, \mathbb{S}_1, \mathbb{B}_1, \mathbb{T}_1 \rangle_{n_1} \\ \quad \wedge \\ \sigma \rightsquigarrow_{\xi_2} \langle \mathbb{G}_2, \mathbb{S}_2, \mathbb{B}_2, \mathbb{T}_2 \rangle_{n_2} \end{array} \right\} \Rightarrow \mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\mathbb{G}}(\mathbb{S}_1 \wedge \mathbb{B}_1) \leftrightarrow \bar{\exists}_{\mathbb{G}}(\mathbb{S}_2 \wedge \mathbb{B}_2)$$

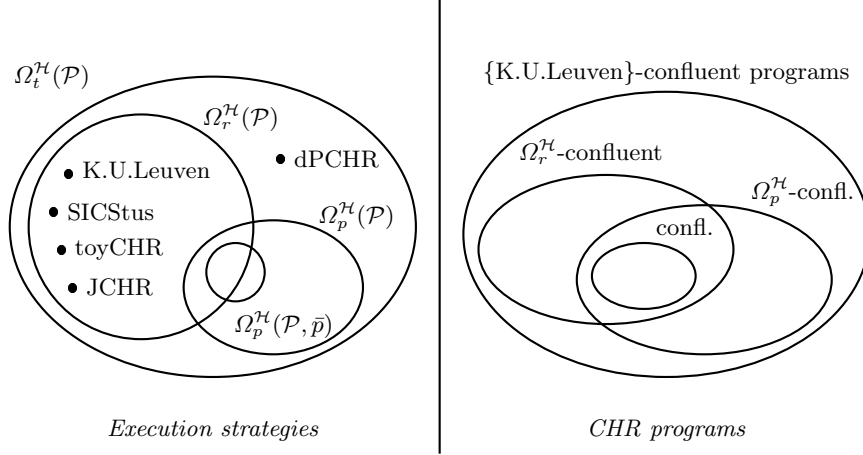


Fig. 2. Execution strategies and CHR programs.

Note that a CHR program \mathcal{P} is $\Omega_t^{\mathcal{H}}(\mathcal{P})$ -confluent if and only if it is confluent (in the usual sense), while $\Omega_r^{\mathcal{H}}(\mathcal{P})$ -confluence corresponds to “confluent in the refined semantics” (see also [2], chapter 6). If the strategy class Ω is a singleton, every CHR program \mathcal{P} is trivially Ω -confluent.

Figure 2 shows some strategy classes and the corresponding sets of Ω -confluent programs. The execution strategy “dPCHR” is that of an implementation of probabilistic CHR [8], in which all rules get the same probability and the rules are picked using a deterministic pseudo-random number generator initialized with some fixed seed.

We have the following duality property: for all strategy classes Ω_1 and Ω_2 : if $\Omega_1 \subseteq \Omega_2$, then every Ω_2 -confluent program is also Ω_1 -confluent.

4 General CHR Machines

We generalize the deterministic abstract CHR machines of Section 2.2 as follows:

Definition 19. A CHR machine is a tuple $\mathcal{M} = (\mathcal{H}, \Omega, \mathcal{P}, \mathcal{V})$ where the host-language \mathcal{H} defines a built-in constraint theory $\mathcal{D}_{\mathcal{H}}$, \mathcal{P} is a CHR program, $\mathcal{V} \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ is a set of valid goals, and $\Omega \subseteq \Omega_t^{\mathcal{H}}(\mathcal{P})$ is a strategy class. The machine takes an input query $\mathbb{G} \in \mathcal{V}$, picks any execution strategy $\xi \in \Omega$, and executes a derivation $d \in \xi|_{\mathbb{G}}$.

Note that we no longer require the program to be $\Delta_{\omega_t}^{\mathcal{H}}$ -deterministic for valid input, and we allow it to use any strategy class.

Terminology. A *CHR(\mathcal{H}) machine* is a CHR machine of the form $(\mathcal{H}, -, -, -)$. An *Ω -CHR machine* is a CHR machine of the form $(-, \Omega, -, -)$. *Abstract CHR*

machines are $\Omega_t^{\mathcal{H}}$ -CHR machines, and *refined* CHR machines are $\Omega_r^{\mathcal{H}}$ -CHR machines. A *feasible* CHR machine is one with a computable strategy class. A *confluent* CHR machine $(-, \Omega, \mathcal{P}, -)$ has a program \mathcal{P} which is Ω -confluent. A *deterministic* CHR machine $(-, \Omega, \mathcal{P}, -)$ has a program \mathcal{P} which is Δ_Ω -deterministic, where $\Delta_\Omega = \bigcup \Omega(\mathcal{P})$ is the set of all possible derivations.

We generalize the definitions of the time and space functions in the straightforward way:

Definition 20. *Given an Ω -CHR machine \mathcal{M} , the time function $chrtime_{\mathcal{M}}$ returns the worst-case derivation length, given an initial goal $\mathbb{G} \in \mathcal{V}\mathcal{G}$:*

$$chrtime_{\mathcal{M}}(\mathbb{G}) = \max\{\#d \mid \exists \xi \in \Omega : d \in \xi|_{\mathbb{G}}\}$$

Definition 21. *Given an Ω -CHR machine \mathcal{M} , the space function $chrspac_{\mathcal{M}}$ returns the worst-case execution state size, given an initial goal:*

$$chrspac_{\mathcal{M}}(\mathbb{G}) = \max\{\text{SIZE}(\sigma) \mid \exists \xi \in \Omega : \sigma \in \xi|_{\mathbb{G}}\}$$

It is not clear what the added power is of generalized CHR machines compared to CHR machines that follow the abstract operational semantics. For well-known strategy classes, like $\Omega_r^{\mathcal{H}}$ or $\Omega_p^{\mathcal{H}}$, we can still only decide the languages in \mathcal{P} in polynomial time. However, it seems that more instantiated strategy classes add some power. For example, in the refined semantics, we can (non-monotonically) check for absence of constraints, and in the priority semantics, we can easily sort in linear time. We can imagine more exotic strategy classes, that could still be computable while implicitly requiring more than a polynomial amount of work to compute the next transition. For such strategy classes, the corresponding generalized CHR machine could of course be much more powerful.

5 Non-deterministic CHR Machines

We define *non-deterministic* CHR machines similarly to the way non-deterministic Turing machines are defined.

Definition 22. *A non-deterministic CHR machine (NCHR machine) is a tuple $\mathcal{M} = (\mathcal{H}, \Omega, \mathcal{P}, \mathcal{V}\mathcal{G})$, where \mathcal{H} , Ω , \mathcal{P} , and $\mathcal{V}\mathcal{G}$ are defined as before. The machine takes an input query $\mathbb{G} \in \mathcal{V}\mathcal{G}$ and considers all execution strategies $\xi \in \Omega$. If there are strategies that result in a successful derivation $d \in \xi|_{\mathbb{G}}$, any of those is returned. Otherwise, any failure derivation is returned. If all derivations are infinite, any infinite derivation is returned.*

As an example, consider the following CHR program $\mathcal{P}_{3\text{SAT}}$:

```

clause(A,_,_) <=> true(A).
clause(_,B,_) <=> true(B).
clause(_,_,C) <=> true(C).
true(X), true(not(X)) <=> fail.

```

The NCHR machine $(\emptyset, \Omega_t^{\mathcal{H}}, \mathcal{P}_{3\text{SAT}}, 3\text{SATCLAUSES})$ decides 3SAT clauses in linear time. A 3SAT clause of the form $(x_1 \vee x_2 \vee x_3) \wedge (y_1 \vee y_2 \vee y_3) \wedge \dots$ is encoded as a query `clause(x1,x2,x3), clause(y1,y2,y3), ...`, where negative literals are encoded by wrapping them in `not/1`. The valid goals `3SATCLAUSES` are all goals of this form.

In every derivation, all `clause/3` constraints are simplified, so one of the literals has been made true. If the instance of 3SAT has a solution, there is a way to do this without producing a conflicting truth assignment, so there is a successful derivation. If there is no solution, all derivations will fail because every assignment causes a conflict, which fires the fourth rule.

The NCHR machine $(\emptyset, \Omega_r^{\mathcal{H}}, \mathcal{P}_{3\text{SAT}}, 3\text{SATCLAUSES})$ — the same as above but with its strategy class restricted to the refined semantics — is no longer correct. Because execution strategies are limited to those of the ω_r semantics, every `clause/3` constraint will be simplified by the first rule. As a result, the only truth assignment that is tried is to make every first literal of all clauses true.

Non-determinism in rule choice is exploited in the program $\mathcal{P}_{3\text{SAT}}$. However, we can easily transfer the non-determinism to the choice of matching partner constraints for an active constraint. For example, the following program, when executed on a refined NCHR machine, decides 3SAT clauses in linear time:

```
clause(A,B,C) <=> d(X,A), d(X,B), d(X,C), c(X).
c(X), d(X,A) <=> true(A).
true(X), true(not(X)) <=> fail.
```

However, as a general rule, the smaller the strategy class, the harder it is to write a correct NCHR program: when there are less sources of non-determinism, the corresponding NCHR machine becomes less powerful. When the strategy class is a singleton, there is of course no difference between a regular CHR machine and an NCHR machine.

For regular CHR machines, the reverse rule of thumb holds: the larger the strategy class, the harder it is to write a correct CHR program — more non-determinism only means more wrong choices. If we denote the class of decision problems that can be solved by a deterministic Ω -CHR machine in polynomial time with P_Ω , and the class of decision problems that can be solved by a polynomial-time non-deterministic Ω -CHR machine with NP_Ω , then we have the following inclusions:

$$P_{\Omega_t^{\mathcal{H}}} \subseteq P_{\Omega_r^{\mathcal{H}}} \subseteq P_{\{\text{K.U.Leuven}\}} = NP_{\{\text{K.U.Leuven}\}} \subseteq NP_{\Omega_r^{\mathcal{H}}} \subseteq NP_{\Omega_t^{\mathcal{H}}}$$

Most of these inclusions collapse to equalities: since the RAM simulator program of [11] is $(\Omega_t^{\mathcal{H}})$ -confluent, we have $P_{\Omega_t^{\mathcal{H}}} = P_{\{\text{K.U.Leuven}\}}$. In this sense, the strategy class does not seem to affect the computational power of the CHR machine. Still, it is our experience that it is easier to write programs for a more instantiated operational semantics. In the case of the self-modifying CHR machines of the next section, it does seem to be the case that instantiating the operational semantics really adds power to the machine.

The class of languages that are decided in polynomial time by non-deterministic CHR machines (refined or abstract) coincides with NP. We prove the inclusion $NP_{\Omega^n} \subseteq NP$:

Theorem 3. *The non-deterministic refined CHR-only machine can simulate a non-deterministic Turing machine with the same complexity.*

Proof. The simulator program TMSIM (Fig. 1 page 4) also works if `delta/5` is not a function but defines more than one transition for a given state and symbol. The non-determinism in choosing the partner constraint for `head/1` (the computation-driving active constraint) ensures that all Turing machine computation paths are simulated.

The simulator program also works in the (non-deterministic) abstract semantics.

6 Self-modifying CHR Machines

The RASP machine (random access stored program) [10] is essentially a RAM machine which can access and change its own program instructions — much like real computers which follow the von Neumann architecture: instructions and data are stored in the same memory space, hence the term *stored program*. In terms of computational power and complexity, the RASP machine is just as powerful as a regular RAM machine; the reason is that you can write a RASP simulator on a RAM machine, which takes only a constant factor more time and space.

In this section, we examine CHR machines with a stored program, or CHRSP machines. Since the CHR program is now stored in the CHR store, it can be accessed and modified like any other CHR constraints.

6.1 Definition

We use a syntax that somewhat resembles that of [7], where it was proposed in the context of source-to-source transformation. A CHR program is encoded using “reserved keyword” constraints `rule/1`, `khead/2`, `rhead/2`, `guard/2`, and `body/2`. For example, the rules

```
foo @ bar ==> baz.
qux @ blarg \ wibble <=> flob | wobble.
```

would be encoded as

```
rule(foo), khead(foo,bar), guard(foo,true), body(foo,baz),
rule(qux), khead(qux,blarg), rhead(qux,wibble),
guard(qux,flob), body(qux,wobble)
```

3. Apply. $\langle \mathbb{G}, H_1 \cup H_2 \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto_{\mathcal{P}} \langle C' \uplus \mathbb{G}, H_1 \cup \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
 where $chr(\mathbb{S})$ contains the following constraints, which encode a rule: $rule(r)$,
 $khead(r, h_1), \dots, khead(r, h_k), rhead(r, h_{k+1}), \dots, rhead(r, h_l)$,
 $guard(r, g)$, $body(r, C)$, and neither \mathbb{G} nor \mathbb{S} contain any other rule-encoding
 constraints with r as a first argument. Now let g', C', H'_1 , and H'_2 be consistently
 renamed apart versions of $g, C, (h_1 \wedge \dots \wedge h_k)$, and $(h_{k+1} \wedge \dots \wedge h_l)$, respectively.
 This encoding corresponds to a rule of the form $r @ H'_1 \setminus H'_2 \iff g' | C'$. As usual,
 θ is a matching substitution such that $chr(H_1) = \theta(H'_1)$ and $chr(H_2) = \theta(H'_2)$
 and $h = (r, id(H_1), id(H_2)) \notin T$ and $\mathcal{D}_{\mathcal{H}} \models (\exists_{\emptyset} \mathbb{B}) \wedge (\mathbb{B} \rightarrow \exists_{\mathbb{B}}(\theta \wedge g'))$.

Fig. 3. The new **Apply** transition in the ω_t^{sp} semantics for CHRSP machines

Now, we modify the **Apply** transition of ω_t to refer to the stored program as in Fig. 3. Note that because the program is now in the constraint store, we no longer need to (implicitly) parametrize the semantics with a CHR program; the program is now part of the query or, equivalently, corresponds to the initial store. In order to avoid premature application of rules, i.e. firing a rule which is still being constructed, we require that a rule can only fire if there are none of its components in the goal, waiting to be introduced into the store.

A CHRSP machine is defined just like a regular CHR machine, except that the operational semantics is altered in the way described above. Also, if a program is given for CHRSP machines, it should be considered to be an abbreviation for the encoded form, which is implicitly appended to all valid goals. As before, every ω_t^{sp} transition takes constant time; the machine rejects the input if the final state is a failure state, otherwise it accepts or does not terminate.

6.2 Complexity of CHRSP Machines

Unlike RASP machines, which can be efficiently simulated on RAM machines, CHRSP machines cannot be simulated on regular CHR machines with only constant overhead. The reason is that finding k partner constraints in a store of size n can take $\mathcal{O}(n^k)$ time. For a fixed program, k is bounded, but on a CHRSP machine, rules with an arbitrary number of heads may be created.

In fact, self-modifying CHR programs can decide co-NP-complete languages in only linear time. Consider the problem of Hamiltonian paths in directed graphs. Deciding whether a graph has a Hamiltonian path is NP-complete; the language of consisting of all graphs that do not have a Hamiltonian path is therefore co-NP-complete. Now consider the following self-modifying CHR program:

```

size(N) <=> rule(find_path), size(N,A).
size(N,A) <=> N>1 |
    khead(find_path,node(A)),
    khead(find_path,edge(A,B)),
    size(N-1,B).
size(1,A) <=>

```

```

khead(find_path,node(A)),
body(find_path,fail).

```

As input query, we encode a graph in the following way: `edge/2` constraints for the edges; `node/1` constraints for the n nodes; one `size(n)` constraint to indicate the number of nodes. The program will create a rule of the form

```

find_path @ node(A1), node(A2), ..., node(An),
edge(A1,A2),edge(A2,A3), ..., edge(An-1,An) ==> fail.

```

If the graph has a Hamiltonian path, this rule fires and the CHRSP machine rejects the input. Otherwise, the machine accepts the input. Either way, the machine halts after $\mathcal{O}(n)$ steps.

If a regular CHR machine exists that can simulate CHRSP machines with only polynomial overhead, then $\text{co-NP} \subseteq \text{P}$, and thus $\text{P} = \text{NP}$. So if $\text{P} \neq \text{NP}$, CHRSP machines are strictly more powerful than regular CHR machines.

7 Summary and Conclusion

We have defined three different generalizations of the CHR machine of [11]: CHR machines with restricted strategy classes, non-deterministic CHR machines, and stored-program (self-modifying) CHR machines. These generalizations are orthogonal, so they can be combined. Indeed, one could very well consider, for instance, a refined self-modifying CHR machine. We have investigated the complexity properties of these generalized CHR machines.

7.1 Complexity Summary

As shown in [11], a regular CHR machine can do in polynomial time what a Turing machine (or a RAM machine) can do in polynomial time:

$$P_{\Omega_t} = P$$

In Section 5 we showed a similar result for non-deterministic CHR machines:

$$NP_{\Omega_t} = NP$$

However, although $P_{\text{RASP}} = P$, self-modifying CHR machines are more powerful than regular ones, although exact bounds are still an open problem:

$$\text{coNP} \subseteq P_{\Omega_t^{\text{sp}}} \subseteq \text{PSPACE}$$

Restricting the strategy class to an instantiation of Ω_t (or Ω_t^{sp}) can make the CHR machine stronger: a self-modifying *refined* CHR machine can also solve NP -complete problems in linear time by checking for absence of a solution to the corresponding coNP problem. Checking for absence is not known to be possible in the abstract semantics. So $P_{\Omega_t^{\text{sp}}} \subseteq P_{\Omega_r^{\text{sp}}}$ (and we conjecture the inclusion to be strict), and also $\text{coNP} \cup \text{NP} \subseteq P_{\Omega_r^{\text{sp}}}$.

7.2 Future Work

Determining the exact complexity classes corresponding to CHRSP machines is still an open problem. It is also not clear to what extent the choice of strategy class influences the computational power, even for regular generalized CHR machines, let alone in the case of non-deterministic and/or stored-program CHR machines.

References

1. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for CHR. In M. Leuschel and A. Podelski, editors, *PPDP '07*, pages 25–36, Wrocław, Poland, July 2007. ACM Press.
2. Gregory J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Victoria, Australia, December 2005.
3. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaaur. The refined operational semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *ICLP '04*, volume 3132 of *LNCS*, pages 90–104, Saint-Malo, France, September 2004. Springer.
4. Gregory J. Duck, Peter J. Stuckey, and Martin Sulzmann. Observable confluence for Constraint Handling Rules. In V. Dahl and I. Niemelä, editors, *ICLP '07*, volume 4670 of *LNCS*, pages 224–239, Porto, Portugal, September 2007. Springer.
5. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
6. Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2008. To appear.
7. Thom Frühwirth and Christian Holzbaaur. Source-to-source transformation for a class of expressive rules. In F. Buccafurri, editor, *AGP '03: Joint Conf. Declarative Programming APPIA-GULP-PRODE*, pages 386–397, Reggio Calabria, Italy, September 2003.
8. Thom Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic Constraint Handling Rules. In M. Comini and M. Falaschi, editors, *WFLP '02: Proc. 11th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, volume 76 of *ENTCS*, Grado, Italy, June 2002. Elsevier.
9. Cinzia Di Giusto, Maurizio Gabbrielli, and Maria Chiara Meo. Expressiveness of multiple heads in CHR. Submitted to *PPDP'08*, 2008.
10. Juris Hartmanis. Computational complexity of random access stored program machines. *Theory of Computing Systems*, 5(3):232–245, September 1971.
11. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. Submitted to *ACM TOPLAS*, 2008.
12. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. Submitted to *Theory and Practice of Logic Programming*, 2008.