The Computational Power and Complexity of Constraint Handling Rules

Jon Sneyers*, Tom Schrijvers**, Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium

Abstract. We introduce the CHR machine, a model of computation based on the Constraint Handling Rules programming language. Its computational power and time complexity properties are compared with those of the Turing machine and Random Access Memory machine. Our major result is the proof that every algorithm can be implemented in CHR with the best known time and space complexity.

1 Introduction

Constraint Handling Rules (CHR) [5] is a high-level programming language extension based on multi-headed committed-choice rules. Originally designed for writing constraint solvers, it is increasingly used for general purposes.

After a new programming language is introduced, sooner or later the question arises whether classical algorithms can be implemented in an efficient and elegant way. In [15] this question is partially answered for CHR by describing a CHR implementation of the union-find algorithm. Thanks to some compiler optimizations, most notably hash-table constraint stores, the best known time complexity was obtained. In contrast, it is not clear whether the union-find algorithm can be implemented with optimal complexity in pure Prolog [7].

The major result of this paper is an affirmative answer to the first part of the above question: *is it possible, in general, to implement algorithms in CHR in an efficient way?* i.e. with the best known time and space complexity. We have investigated this question by introducing a new model of computation, the CHR machine, and comparing it with the Turing machine and RAM machine models.

[6] gives a complexity meta-theorem for CHR programs with only simplification rules. It provides a general technique for obtaining (crude) upper bounds on the time complexity of such programs. Two rather orthogonal components need to be considered in this analysis: the number of rule applications in a derivation, and the cost of finding applicable rules. Considering the current state-of-the-art in CHR compilation, the former component depends only on the CHR program itself, while the latter is implementation-dependent. In an attempt to separate these two components, we will assume the CHR machine to be able to find applicable rules in constant time. This assumption is realistic for a class of CHR

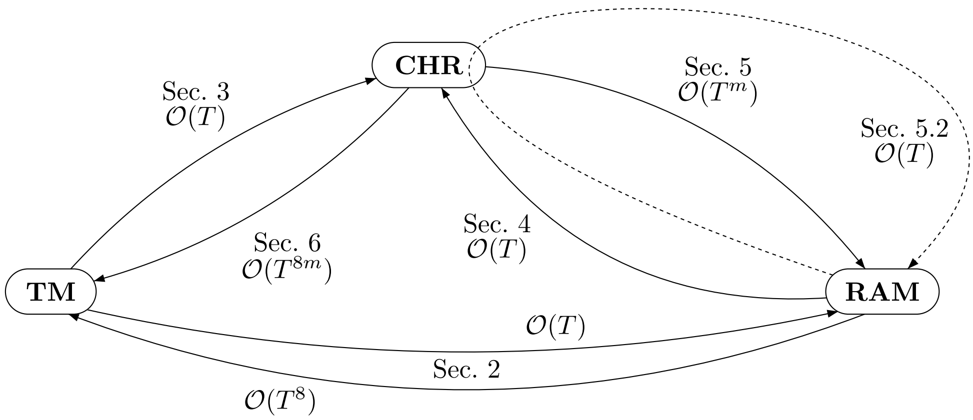
* This work was partly supported by project G.0144.03 funded by F.W.O.-Vlaanderen.

** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

programs, including the simulators we will describe, but it is not feasible in the general case. New compiler optimizations allow a more efficient search for applicable rules. In our terminology, such optimizations correspond to better RAM machine simulators of the CHR machine.

Overview. In the next section we define Turing machines and RAM machines, because multiple definitions are used in the literature, and we introduce the CHR machine. We discuss simulation of Turing machines and RAM machines by CHR in Sections 3 and 4. Then, in Section 5, we investigate RAM machine simulation of CHR. From these results, we derive a theoretical simulation of CHR by a Turing machine in Section 6. Finally, in Section 7 we conclude this paper.

The following graph illustrates the global structure of this paper, and summarizes the complexity results. An arrow from computation model X to computation model Y means that X can be simulated in Y. The arrows are annotated with a reference to the relevant section, and an upper bound on the complexity penalty for the simulation, where T represents the original time complexity and m is the maximal number of head constraints in a rule of the CHR program.



2 Models of computation

In this section, we will describe three models of computation: the Turing machine, the Random Access Memory (RAM) machine, and the Constraint Handling Rules (CHR) machine. We discuss two variants of the RAM machine with a different instruction set: the Peano-arithmetic RAM machine and the standard-arithmetic RAM machine. Similarly, we define two variants of the CHR machine: the CHR-only machine and the minimal host-language CHR machine.

2.1 Turing machines

We will use the single-tape formulation of the Turing machine (TM). Multi-tape Turing machines can be simulated on single-tape Turing machines with a quadratic complexity penalty [8]. The definition below corresponds to [11].

Definition 1 (Turing Machine). *A Turing machine is defined as a 6-tuple* $M = \langle Q, \Sigma, s_0, b, F, \delta \rangle$ *where*

- Q *is a finite set of* states
- Σ *is a finite set of symbols, the tape* alphabet
- $s_0 \in Q$ *is the initial state*
- $b \in \Sigma$ *is the* blank *symbol*
- $F \subseteq Q$ *is the set of* accepting *final states*
- $\delta : Q \setminus F \times \Sigma \mapsto Q \times \Sigma \times \{L, R\}$ *is a partial function called the* transition function *where* L *is left shift,* R *is right shift*

In the literature definitions sometimes differ, but without an impact on the computational power or time complexity. For example the set $\{L, R\}$ is often extended with S, allowing the machine to stay on the same tape cell.

A Turing machine M operates on an infinite tape of cells. Each cell contains a symbol of the tape alphabet Σ. The tape is assumed to be arbitrarily extendible to the left and the right. A head is positioned on a particular cell of the tape, can read and write a symbol in that cell and can move left and right.

Operation starts in the initial state s_0 on a tape with a finite number of non-blank cells (called the *input*) and the head is positioned on the left-most non-blank symbol. Execution proceeds by considering the current state s and the symbol σ that is under the head. Then:

- Either (s, σ) is a member of the domain of δ and $\delta(s, \sigma) = (s', \sigma', X)$. The effect then is that the current state of M changes to s', the head overwrites the value σ in the cell under it with σ' and next the head either moves to the left or the right depending on whether $X = L$ or $X = R$.
- Or (s, σ) is not part of the domain of δ. Execution stops. If $s \in F$, the input is *accepted*, otherwise it is *rejected*.

Time complexity. The (worst-case) time complexity of a Turing machine M is the maximal number of execution steps, or state changes, as a function of the size of the input. The size of the input is usually defined as the number of symbols between the left-most and right-most non-blank symbols.

Turing-completeness. A programming language is *Turing-complete* if it has the same computational power as Turing machines. In other words, every Turing Machine can be simulated in the programming language and every program of the language can be simulated on a Turing machine.

2.2 Random Access Memory machines

The Random Access Memory (RAM) machine closely models the basic features of traditional sequential computers. In the literature many variations of the RAM have been considered. We investigate two different RAM machines. The first is a RAM machine with simple Peano arithmetic operations and the second has the standard arithmetic operations as they are implemented on today's computers.

Instruction		*Effect*
inc	A	Increment the value of register A by one.
dec	A	Decrement the value of register A by one.
clr	A	Set the value of register A to zero.
const	$B\ \ A_1$	Set the value of register A_1 to the value B.
add	$A_2\ A_1$	Add the value of register A_2 to the value of register A_1.
sub	$A_2\ A_1$	Subtract the value of register A_2 from the value of register A_1.
mult	$A_2\ A_1$	Multiply the value of register A_2 to the value of register A_1.
div	$A_2\ A_1$	Divide the value of register A_1 by the value of register A_2.
move	$A_2\ A_1$	Set the value of register A_1 to the value of register A_2.
i_move	$A_2\ A_1$	Set the value of register A_1 to the value of the register given by the value of register A_2.
move_i	$A_2\ A_1$	Set the value of the register given by the value of register A_1 to the value of register A_2.
jump	L	Set the program counter to L.
cjump	$A\ \ L$	If the content of register A is zero, set the program counter to L; otherwise continue.
halt		Halt execution of the RAM.

Table 1. Instruction set of the RAM

Common architecture. A RAM machine consists of two components: the central processing unit (CPU) and a random-access memory. The memory consists of a number of cells, or registers. Every cell has an address and a value. Addresses are successive natural numbers. In our models the memory is split into two disjoint parts with separate address spaces: the first is the working memory whose cells contain integers and the second is the program memory whose cells contain the successive instructions of the program. The working memory is initialized appropriately, i.e. the values of all the memory cells are zero except those that contain the input parameters. By convention, particular memory cells may contain the output of an execution. We use $A, A_1, A_2, \ldots$ to represent working memory addresses and $L, L_1, \ldots$ for program memory addresses.

The CPU follows a fetch-and-execute cycle. It has a program counter that is initialized to the first program memory address. This program pointer contains the address of the program memory cell with the next instruction to be executed. The CPU fetches the instruction in the cell and performs the corresponding operations. This involves setting the program counter to the next instruction, by default the successor of the current address. Table 1 lists the instructions supported by the standard-arithmetic RAM machine. The Peano-arithmetic RAM machine uses a subset of these instructions.

Definition 2 (Peano-Arithmetic RAM). *A Peano-arithmetic RAM machine consists of a program and a working memory as described above. The program instructions are **inc**, **dec**, **clr**, **jump**, **cjump**, and **halt** (see Table 1).*

This corresponds to the definition given in [12]. All copying, addition and subtraction has to be done by repeated use of the **inc** and **dec** instructions. This makes the Peano-arithmetic RAM less realistic, as actual computers do provide instructions for addition and subtraction. However, pure CHR without built-in constraints does not provide any arithmetic functionality either. Hence, arithmetic must be encoded, for example using Peano arithmetic.

Definition 3 (Standard-Arithmetic RAM). *A standard-arithmetic RAM machine (or standard RAM) consists of a program and a working memory as described above. The program instruction set is given in Table 1.*

This instruction set is similar to the one defined in [3], and resembles more closely actual computers. The **inc**, **dec**, and **clr** instructions are redundant: they can be implemented using **add**, **sub**, and **const**.

Time complexity. Every fetch-and-execute cycle in the RAM machine comprises one step. For this to be realistic, the memory cells have to be of bounded size. Without upper bound, a standard RAM can compute, in $\mathcal{O}(n)$ steps and without input, numbers as high as 2^{2^n}, which require $\mathcal{O}(2^n)$ Turing machine steps just to write to the tape. This means that a polynomial relationship between standard RAM machines and Turing machines would be impossible. Addition and subtraction take linear time in the Peano-arithmetic RAM, whereas they happen in constant time in the standard RAM.

Turing-completeness of the RAM. The two formulations of the RAM machine are both Turing-complete. For both RAM machines, a T-time Turing machine can be simulated in $\mathcal{O}(T)$ time. The main difference between the two RAM machines is in the effect on time complexity when they are simulated on a Turing machine. In [12] it is reported that a T-time Peano-arithmetic RAM using S registers can be simulated on a Turing machine in $\mathcal{O}(ST \log^2 S)$ time. The standard RAM is also polynomially related to the Turing machine, although the complexity penalty of simulating a it on a TM is much higher. It can be shown (see e.g. [3]) that a standard RAM M with time complexity T can be simulated on a multi-tape TM with $\mathcal{O}(T^4)$ time complexity[1]. Because simulating a multi-tape TM on a single-tape TM squares the time complexity [8], we have:

Lemma 1. *Any standard-arithmetic RAM machine with time complexity T can be simulated on a Turing machine with time complexity $\mathcal{O}(T^8)$.*

2.3 CHR machines

We assume the reader to be familiar with CHR. See [5] for an overview of the Constraint Handling Rules (CHR) language. We say a CHR program is *CHR-only* (this term was coined in [14]) if its rules do not contain any host-language

[1] Actually, a tighter bounds of $\mathcal{O}\big((T \log T \log\log T)^2\big)$ may be shown, as mentioned in a footnote on page 26 of [3].

built-ins in the body, and contain only syntactic (in)equalities in the guard. Furthermore, no compound Prolog terms are allowed: constraint arguments are constants or variables.

If we assume the existence of constant-time built-in arithmetic operations (one-way constraints) provided by the host language, then CHR can match the arithmetic efficiency of the standard RAM. This assumption is not unrealistic as the host language of CHR is often a language implemented on a standard RAM. For example, the host language Prolog provides the `is/2` one-way constraint that implements the arithmetic operations through delegation of the corresponding operations to the RAM on which Prolog is interpreted. Therefore, it makes sense to look at CHR programs in which statements are allowed that are likely to be available in every host-language: the basic arithmetic and comparison operations. We call such programs *minimal host-language CHR programs*. The set of extra built-in constraints must contain at least $+$, $-$, $*$, $/$, $=$, and $\neq$.

Operational semantics. We will use the standard operational semantics of CHR, sometimes also called *abstract* or *high-level* operational semantics. Several versions of the operational semantics have already appeared in the literature, e.g. [1, 5]. We adopt the version of [4], which is equivalent to the previous ones, but closer in formulation to the refined semantics, used in most implementations. The operational semantics is formulated as a state transition system.

Execution state. Firstly, we define an *execution state* σ as a tuple $\langle G, S, B, T\rangle_n$. The first part of the tuple, the *goal* G, is the multiset of constraints to be rewritten to solved form. The CHR constraint *store* S is the multiset of *identified* CHR constraints that can be matched with rules in the program P. An *identified* CHR constraint is a CHR constraint c associated with some unique integer i, which serves to differentiate among copies of the same constraint. We introduce the functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences, sets and multisets of identified CHR constraints in the obvious manner. The constraint theory of the built-in constraints is denoted by $\mathcal{D}_b$.

The *built-in constraint store* B is the conjunction of all built-in constraints that have been passed to the underlying solver. For CHR-only programs, the built-in store is not needed. Since we will usually have no information about the internal representation of B, we will model it as a logical conjunction of constraints. The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. This is necessary to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints have not been used (in the same order) to fire the same rule before. Finally, the counter n represents the next free integer that can be used to number a CHR constraint. Given an input query Q, the *initial execution state* is: $\langle Q, \emptyset, true, \emptyset\rangle_1$.

Transition rules. The standard operational semantics is based on the three transition rules listed in Figure 1. The first rule tells the underlying solver to

1. Solve $\langle \{c\} \uplus G, S, B, T\rangle_n \rightarrowtail_{solve} \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint. This transition does not occur in CHR-only programs.

---

2. Introduce $\langle \{c\} \uplus G, S, B, T\rangle_n \rightarrowtail_{introduce} \langle G, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$ where c is a CHR constraint.

---

3. Apply $\langle G, H_1 \uplus H_2 \uplus S, B, T\rangle_n \rightarrowtail_{apply} \langle C \uplus G, H_1 \uplus S, \theta \wedge B, T \cup \{h\}\rangle_n$ where there exists a (renamed apart) rule in $\mathcal{P}$ of the form

$$r \ @ \ H_1' \setminus H_2' \iff g \mid C$$

and a matching substitution θ such that $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$ and $\mathcal{D}_b \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$ and $h = (r, id(H_1), id(H_2)) \notin T$.

Fig. 1. The transition rules of the standard operational semantics

add a new built-in constraint to the built-in constraint store. Of course, this transition rule is not relevant for CHR-only programs. The second adds a new identified CHR constraint to the CHR constraint store. The last one chooses a program rule for which matching constraints exist in the CHR constraint store, and whose guard is entailed by the underlying solver, and *fires* it.

The transitions are non-deterministically applied, starting from the initial execution state, until either no more transitions are applicable (a successful derivation), or the underlying solver can prove $\mathcal{D}_b \models \neg\bar{\exists}_\emptyset B$ (a failed derivation). In both cases a *final state* has been reached, the former a *success state* and the latter a *failed state*. CHR-only programs do not have failed derivations.

Definition 4 (CHR-only Machine). *A* CHR-only machine *consists of a CHR-only program and an execution state. Its initial state is given by an* input query, *which is a sequence of constraints. The machine works by exhaustively applying the transitions of the operational semantics, given in Figure 1. The output, or* solved form, *is given by the remaining constraints in the constraint store.*

Definition 5 (Minimal Host-language CHR Machine). *A* minimal host-language CHR machine *consists of a minimal host-language CHR program and an execution state. Input, processing, and output are as in definition 4.*

Clearly, the CHR-only machines are a subset of minimal host-language CHR machines. Both machines are theoretical models because we will assume they can do every transition in constant time.

Time complexity. The time complexity of a particular CHR derivation from an initial to a final state is the number of derivation steps. For a given CHR program P, the time complexity $f_P(q)$ of an initial query q is the maximum length of all derivations starting from q, or $+\infty$ if one of these derivation does not terminate. The size $|q|$ of a query q is the number of constraints in the query.

Definition 6. *The time complexity of a CHR program P is a function* $\mathrm{time}_P(n)$*:*

$$\mathrm{time}_P(n) = \max_{|q|=n} f_P(q)$$

Example. This definition does not correspond to the reality of CHR implementations. For example, a CHR-only machine can solve a crossword puzzle with time complexity linear in the number of words, which is impossible in implementations on a RAM machine. Given a query containing n words from a dictionary as `word/`k constraints (word length k), the following program returns all s solutions for the given crossword puzzle in time $n + s$.

A	B	C	D	E
F		G		H
I	J	K		L
	M	N	O	P
Q	R	S	T	U

Crossword puzzle solver

```
word(A,B,C,D,E), word(I,J,K), word(M,N,O,P),
word(Q,R,S,T,U), word(A,F,I), word(J,M,R),
word(C,G,K,N,S), word(O,T), word(E,H,L,P,U)
    ==>
sol(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U).
```

3 Simulating Turing machines in CHR

Lemma 2. *Any Turing machine can be simulated on a CHR-only machine.*

Proof. Consider the following Turing machine simulator written in (pure) CHR:

Turing Machine Simulator

```
trans @ delta(S,G,Sp,Gp,Dir) \  current_state(S), head(Cell),
                                cell(Cell,G,Left,Right)
                            <=> current_state(Sp),
                                cell(Cell,Gp,Left,Right),
                                move(Dir,Cell,Left,Right).

move(l,Cell,L,_) <=> L \= null | head(L).
move(l,Cell,null,R), cell(Cell,G,null,R)
     <=> cell(Cell,G,L,R), cell(L,b,null,Cell), head(L).
move(r,Cell,_,R) <=> R \= null | head(R).
move(r,Cell,L,null), cell(Cell,G,_,_)
     <=> cell(Cell,G,L,R), cell(R,b,Cell,null), head(R).
```

The meaning of the different constraints in the simulator is:

- `delta/5` encodes the partial transition function in the obvious way. The first two arguments are the inputs and the last three the output of δ.
- `current_state/1` contains the current state.
- `head/1` contains the identifier of the cell under the head.

– `cell/4` represents a cell of the tape. The first argument is the unique identifier of the cell. The second is the symbol in the cell (`b` represents the blank symbol) and the last two are the identifiers of the neighboring cells.
– `move/4` represents a move of the head. The first argument is the direction of movement, `l` for left and `r` for right. The second argument is the identifier of the current cell and the last two are the identifiers of respectively the cell to the left and the right.

The special cell identifier `null` is used to refer to a not yet instantiated cell. The corresponding rules for `move/2` take care of extending the tape.

A simulation of the execution of a Turing machine M proceeds as follows. The the left-most and rightmost non-blank cells on the initial tape are encoded as `cell/4` constraints together with all the cells in between. The identifier of the cell to the left of the left-most non-blank is set to `null` and similarly for the cell to the right of the right-most non-blank cell. 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 `current_state(S0)` where `S0` is the initial state of M and the constraint `head(LeftMost)` where `LeftMost` is the identifier of the left-most non-blank cell.

The reader can easily verify that this CHR program can indeed simulate any Turing machine. Also, the mapping of Turing machine programs and tape inputs to the corresponding query for the simulator is obvious. Finally it is straightforward to extract the output of the simulated Turing machine from the output of the simulator: it suffices to look at the remaining `cell/4` constraints.

3.1 Time complexity

Theorem 1. *A Turing machine with time complexity $\mathcal{O}(T)$ can be simulated on the above CHR machine in time $\mathcal{O}(T)$.*

Proof. If T is the number of steps of the execution of the Turing Machine M, then the number of CHR transition steps in the simulator is bounded by $6T + 2E$. Every step of the Turing machine corresponds with one application of the `trans` rule and one application of either of the move rules. These two rules introduce at least 4 new constraints that need to be inserted. That accounts for the term $6T$. Finally, each time the tape is extended to the left or the right, two additional `cell/4` constraints are inserted by a move rule. E is the number of these extensions, with $E \leq T$.

Hence the time complexity of the simulation of M in CHR is $\mathcal{O}(T)$ where T is the time complexity of M. □

Simulator specialization. The simulator program may be specialized for a particular Turing Machine. Instead of the `trans` rule, the specialized version has one rule of the form:

```
     S, head(Cell), cell(Cell,G,Left,Right)
 <=> Sp, cell(Gp,Left,Right), move(Dir,Cell,Left,Right).
```

for every delta(S,G,Sp,Gp,Dir) constraint. This specialization does not affect the time complexity in an essential way.

4 Simulating RAM machines in CHR

Lemma 3. *Any Peano-arithmetic RAM machine can be simulated on a CHR-only machine, and any standard-arithmetic RAM machine can be simulated on a minimal host-language CHR machine.*

4.1 Simulating the Peano-arithmetic RAM

The following CHR-only program simulates the Peano-arithmetic RAM machine:

```
Peano-arithmetic RAM simulator
prog(L,L1,inc,A) \ m(A,X), pc(L) <=> m(A,Z), s(Z,X), pc(L1).
prog(L,L1,dec,A) \ m(A,X), s(X,Y), pc(L) <=> m(A,Y), pc(L1).
prog(L,L1,clr,A) \ m(A,X), pc(L) <=> m(A,zero), pc(L1).

prog(L,L1,jump,A) \ pc(L) <=> pc(A).
prog(L,L1,cjump,A,L2), m(A,zero) \ pc(L) <=> pc(L2).
prog(L,L1,cjump,A,L2), m(A,X), s(X,_) \ pc(L) <=> pc(L1).

prog(L,L1,halt) \ pc(L) <=> true.
```

The query consists of an encoding of the RAM program as prog/{3,4,5} constraints; the first argument represents the label (or line number), the second argument is the label of the next program line, the third argument is an instruction mnemonic. The other arguments of the prog/{4,5} constraints contain the operands of the instruction. The memory content is represented as a chain of s/2 constraints ending in zero, where s/2 represents the successor function. The memory has to be initialized by adding to the query one m/2 constraint for every register which is used in the program. Finally, the program counter must be set to the label of the first line of the program by adding an initial pc/1 constraint. Note that this CHR program does not rely on numbers or arithmetic operations.

Time complexity. After the initialization, one rule is applied for every step of the Peano-Arithmetic RAM. Applying a rule causes at most three constraints to be inserted. Therefore, the number of CHR steps is bounded by four times the number of RAM steps. Hence, the time complexity of simulating a PA-RAM machine M in CHR(-only) is $\mathcal{O}(T)$ where M has time complexity T.

Initializing the n memory cells to values v_i $(0 \leq i \leq n)$ means inserting n m/2 constraints and m s/2 constraints, with $m = \sum_{i=1}^{n} v_i$.

Theorem 2. *A Peano-arithmetic RAM machine with time complexity $\mathcal{O}(T)$ can be simulated on the above CHR-only machine in time $\mathcal{O}(T)$, not counting memory initialization.*

4.2 Simulating the standard-arithmetic RAM

We can also simulate the more realistic standard-arithmetic RAM machine in CHR. However, if we want to do this without a complexity penalty, we need minimal host language operations to handle the integers. The following minimal host-language CHR program simulates the standard RAM machine. We used Prolog syntax here – but of course the five host-language statements can easily be modified to the syntax of other host-languages like Java.

```
Standard RAM simulator
prog(L,L1,const,B,A) \ m(A,_), pc(L) <=> m(A,B), pc(L1).
prog(L,L1,add,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X+Y, m(A,Z), pc(L1).
prog(L,L1,sub,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X-Y, m(A,Z), pc(L1).
prog(L,L1,mult,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X*Y, m(A,Z), pc(L1).
prog(L,L1,div,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X//Y, m(A,Z), pc(L1).

prog(L,L1,move,B,A), m(B,X) \ m(A,_), pc(L) <=> m(A,X), pc(L1).
prog(L,L1,i_move,B,A), m(B,C), m(C,X) \ m(A,_), pc(L) <=>
    m(A,X), pc(L1).
prog(L,L1,move_i,B,A), m(B,X), m(A,C) \ m(C,_), pc(L) <=>
    m(C,X), pc(L1).

prog(L,L1,jump,A) \ pc(L) <=> pc(A).
prog(L,L1,cjump,R,A), m(R,0) \ pc(L) <=> pc(A).
prog(L,L1,cjump,R,A), m(R,X) \ pc(L) <=> X =\= 0 | pc(L1).

prog(L,L1,halt) \ pc(L) <=> true.
```

Time complexity. The time complexity of this simulator is similar to the Peano-arithmetic RAM simulator. A minor difference is in the time for memory initialization, which does not depend on the size of the input numbers as in the Peano-arithmetic RAM simulator.

Theorem 3. *A standard-arithmetic RAM machine with time complexity $\mathcal{O}(T)$ can be simulated on the above minimal host-language CHR machine in time $\mathcal{O}(T)$, not counting memory initialization.*

5 Simulating CHR on RAM machines

For the simulation of CHR on RAM machines we can refer to the available CHR implementations in various host languages: Prolog [13, 9], HAL [10], Haskell [16] and Java [2, 17]. As all these host languages can be simulated on RAM machines, the simulation of CHR on RAM machines follows.

Theorem 4. *Both CHR machines are Turing-complete.*

Proof. Because of Lemma 2, every Turing machine can be simulated on a CHR-only machine (which is also a minimal host-language CHR machine). Every CHR machine can be simulated on a Turing machine, because every CHR machine can be simulated on a RAM machine (we can use any CHR compiler), and because RAM machines can be simulated on a Turing machine (Lemma 1). □

Part of the non-determinism in the operational semantics of CHR is reduced in the above CHR systems, since they use specific instantiations of the abstract operational semantics. We consider the particular instantiation called the *refined operational semantics* [4], which is used in all major optimizing CHR systems.

5.1 Time complexity of simulating CHR on a RAM machine

The following general result holds for CHR-only programs and for CHR programs where every host language statement takes constant time.

Theorem 5. *A CHR program with time complexity $\mathcal{O}(f(n))$ can be simulated on a RAM machine with time complexity $\mathcal{O}(f(n)^m)$, where m is the maximal number of head constraints in a rule of the CHR program.*

Proof sketch. We will use the execution strategy of the refined operational semantics: after every **Introduce** transition, we try to apply, in textual order, the rules containing the introduced (or *active*) constraint in the head. The **Solve** transition takes constant time by assumption. The **Introduce** transition can also be performed in constant time, given a sufficiently efficient implementation of the constraint store – a linked list suffices. Trying an **Apply** transition can be done as follows: for every occurrence, find the partner constraints in the store and check the guard condition. In the worst case, this takes $\mathcal{O}(RS^{m-1})$ RAM cycles, where R is the number of occurrences of the active constraint and S is the number of constraints in the store. Because S is bounded by $\mathcal{O}(f(n))$, and R is bounded by a constant for a fixed CHR program, we get a worst-case time complexity of $\mathcal{O}(f(n)^{m-1})$ RAM cycles per CHR transition. Hence, the time for simulating $\mathcal{O}(f(n))$ CHR steps on a RAM machine is $\mathcal{O}(f(n)^m)$. □

Note that restricting the execution strategy to the refined operational semantics results in a tighter bound for the cost of trying an **Apply** transition. Lemma 4.1 in [6] gives a complexity of $\mathcal{O}(rS^m)$ where r is the number of CHR rules, which is weaker than our bound of $\mathcal{O}(RS^{m-1})$. The reason is that in the refined semantics not all possible combinations of constraints are tried: only the combinations containing the active constraint are considered.

5.2 RAM → CHR → RAM

We are now ready to state the most important result of this paper. It implies that every algorithm can be implemented in CHR with the best known time and space complexity.

Theorem 6. *For every RAM program P with time complexity $f(n)$, using $g(n)$ memory cells, a minimal host-language CHR program P' exists which computes output corresponding to the output of P. Furthermore, P' can be simulated on a RAM machine with time complexity $\mathcal{O}(f(n))$, using $\mathcal{O}(g(n))$ memory cells.*

Proof sketch. It suffices to show that for the RAM simulator described in Section 4.2, the host-language code generated by the CHR compiler can perform every RAM instruction in constant time, and the host-language code can be executed using $\mathcal{O}(g(n))$ memory. This is in fact the case for the K.U.Leuven CHR compiler [13], which uses intelligent scheduling of partner constraint lookups, and hash-table constraint stores for ground constraints. These features allow a constant-time lookup of the partner `prog/4` and `m/2` constraints since they are functionally dependent on the given `pc/1` constraint. Also, the insertion and deletion of `m/2` constraints in the store can be performed in constant time. Hence, the time complexity of the resulting RAM simulation of P' is linear in the number of rule applications (which is exactly $f(n)$).

Thanks to compile-time garbage collection, space can be reused every time a memory cell constraint `m(A,_)` is updated. Also, potential stack related additional space use can be reduced to a constant because of tail recursion optimization. The propagation history is only needed for propagation rules, which do not occur in the simulator program. Because of the above optimizations, the space usage will be linear in the number of `m/2` constraints (which is exactly $g(n)$). □

This means that if some algorithm can be implemented efficiently in some programming language, it can be implemented in CHR too, with the same time and space complexity. In other words, everything can be implemented in CHR with the best known complexity. Of course, in the worst case this can only be done by simulating the RAM version of an existing program. However, we are convinced that in practice much more elegant CHR programs can be constructed.

6 Simulating CHR on Turing machines

In the previous section, we showed that a CHR program with time complexity $\mathcal{O}(f(n))$ can be simulated on a RAM machine with time complexity $\mathcal{O}(f(n)^m)$. According to Lemma 1, RAM machines with time complexity T can be simulated on a Turing machine with time complexity $\mathcal{O}(T^8)$. Combining these upper bounds, we get the following (expected) result:

Theorem 7. *Any CHR machine with time complexity $\mathcal{O}(f(n))$ can be simulated on a Turing machine with time complexity $\mathcal{O}(f(n)^{8m})$, where m is the maximal number of head constraints in the rules of the CHR machine.*

Corollary 1. *Let* $\mathbf{P} = \mathbf{P}_{\mathrm{TM}}$ *be the class of problems that can be solved with a Turing machine in polynomial time, and* $\mathbf{P}_{\mathrm{CHR}}$ *the class of problems that can be solved with a CHR-only machine in polynomial time. We have* $\mathbf{P}_{\mathrm{TM}} = \mathbf{P}_{\mathrm{CHR}}$.

Proof. Follows from Theorem 1 and Theorem 7. □

7 Conclusion

We have generalized the optimal complexity result obtained for the union-find algorithm [15] to any algorithm which can be executed on a RAM machine. In other words, we showed that every algorithm can be implemented in CHR with the best-known time and space complexity.

We have shown that the CHR language is Turing-complete. We have introduced a new model of computation, the CHR machine. We have established upper bounds on the penalties for simulating Turing machines and RAM machines on a CHR machine, and vice versa, and found that all three models are polynomially related. This was done by providing and analyzing Turing machine and RAM machine simulators written in CHR.

Future work. An open problem is to formulate, if possible, the equivalent of the linear speedup theorem [8] for CHR machines. This would involve combining derivation steps by combining rules, improving the time complexity by a constant factor. It is unlikely that such a theorem would be useful in practice.

Another open question, related to database indexing, is a syntactic characterization of CHR programs that admit finding applicable rules in constant time.

There seems to be a similarity between non-deterministic Turing machines and CHR programs which allow multiple execution paths. For confluent programs, some paths will lead more quickly to a solution than others, and depending on the execution strategy, the program might have a different time complexity. Non-confluent programs are of course even more dependent on the specifics of the execution strategy. A non-deterministic Turing machine could be modelled as a non-confluent CHR program, where the non-deterministic choices correspond to execution strategy decisions.

Another almost unexplored topic is the study of what could be called a *k-parallel CHR machine* ($k \in \mathbb{N}_0 \cup \{\infty\}$) : a CHR machine able to perform at most k transitions simultaneously, as long as the multisets of removed constraints of every rule application are disjunct.

A variant of the RAM machine, the RASP machine (Random Access Stored Program), has its program in the working memory, allowing self-modifying programs. Both models are equivalent within a constant factor. The concept of self-modifying CHR programs needs to be investigated. Interestingly, self-modifying CHR programs are not polynomially related to the models discussed in this paper, since the number of head constraints in a rule would not remain constant for a given program.

References

1. Slim Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In Gert Smolka, editor, *CP'97: Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, pages 252–266, Schloss Hagenberg, Austria, 1997. Springer Verlag.
2. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A Java Constraint Kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming*, Kiel, Germany, September 2001.
3. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addisson-Wesley, 1975.
4. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, St-Malo, France, September 2004. Springer Verlag.
5. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
6. Thom Frühwirth. As Time Goes By: Automatic Complexity Analysis of Concurrent Rule Programs. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, April 2002.
7. Harald Ganzinger and David McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proceedings of the First International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528, Siena, Italy, June 2001. Springer Verlag.
8. Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Trans. American Mathematical Society*, 117:285–306, May 1965.
9. Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In Gopalan Nadathur, editor, *Proceedings of the Intl. Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 117–133. Springer Verlag, 1999.
10. Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming, Special Issue on CHR*, 5(4+5), 2005.
11. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Longman, 2001.
12. John E. Savage. *Models of Computation*. Addison-Wesley, 1998.
13. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, Ulm, Germany, May 2004.
14. Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. Automatic Implication Checking for CHR Constraint Solvers. Report CW 402, K.U.Leuven, Department of Computer Science, Leuven, Belgium, January 2005.
15. Tom Schrijvers and Thom Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 2005. To appear.
16. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transations on Programming Languages and Systems*, 2005. To appear.
17. Armin Wolf. Adaptive Constraint Handling with CHR in Java. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 2239. Springer Verlag, 2001.