Transforming Imperative Algorithms to Constraint Handling Rules

Slim Abdennadher, Haythem Ismail, and Frederick Khoury

Department of Computer Science, German University in Cairo
[slim.abdennadher, haythem.ismail, frederick.nabil]@guc.edu.eg
http://met.guc.edu.eg

Abstract. Different properties of programs, implemented in Constraint Handling Rules (CHR), have already been investigated. Proving these properties in CHR is fairly simpler than proving them in any type of imperative programming language, which triggered the proposal of a methodology to map imperative programs into equivalent CHR. The equivalence of both programs implies that if a property is satisfied for one, then it is satisfied for the other.
The mapping methodology could be put to other beneficial uses. One such use is the automatic generation of global constraints, at an attempt to demonstrate the benefits of having a rule-based implementation for constraint solvers.

1 Introduction

Algorithms have properties that define their operation and their results, such as correctness and confluence which can be illustrated and proven in programs of various languages. Due to the differences between languages, proofs differ from one to another, and could therefore be easier in some than in others. The aim of the paper is to present an approach to map imperative programs to equivalent rule-based ones. Thus, a technique to prove a property in an imperative program would be to prove this same property in the corresponding rule-based version.
In this paper, a mapping from an imperative programming language to Constraint Handling Rules (CHR) is presented. CHR is a concurrent, committed-choice constraint logic programming language especially designed to implement constraint solvers. CHR, which was developed as an enhancement to the constraint programming paradigm, is declarative and rule-based, and is the language of choice in this paper due to the already existing results to prove several properties, such as correctness, confluence, and termination.
Other areas of research could also benefit from the previously mentioned transformation, such as the development of rule-based solvers for global constraints. The mapping schematic in this work could be applied in the translation of imperative constraint solvers to declarative rule-based solvers, with a purpose of analyzing these constraints through CHR. With the flexibility and expressivity of CHR, such a translation could have benefits on the functionality of the involved constraint-handlers.

This paper is structured as follows. In Section 2 we briefly present the syntax and semantics of a subset of CHR. In Section 3, we give a formal presentation of the mapping and prove the equivalence of the imperative algorithm and the corresponding generated CHR program. In Section 4, we present using examples the methodology for transforming imperative algorithms into CHR. Finally, we conclude in Section 5 with a summary and a discussion of future work.

2 Constraint Handling Rules

This section presents the syntax and the operational semantics of a subset of CHR, namely simpagation rules. We use two disjoint sorts of predicate symbols for two different classes of constraints: *built-in constraint symbols* and *user-defined constraint symbols (CHR symbols)*. Built-in constraints are those handled by a predefined constraint solver that already exists. User-defined constraints are those defined by a CHR program. Simpagation rules are of the form

$$\mathit{Rulename} \ @\ H_1 \backslash H_2 \ \texttt{<=>}\ C \ \texttt{|}\ B,$$

where *Rulename* is an optional unique identifier of a rule. The *head* $H_1 \backslash H_2$ consists of two parts H_1 and H_2. Both parts consist of a conjunction of user-defined constraints. The *guard* C is a conjunction of built-in constraints, and the *body* B is a conjunction of built-in and user-defined constraints. If H_1 is an empty conjunction, then we omit the symbol "\" and the rule is called a simplification rule.
The operational semantics of a simpagation rule is based on an underlying theory CT for the built-in constraints and a state G which is a conjunction of built-in and user-defined constraints. A simpagation rule of the form $H_1 \backslash H_2 \Leftrightarrow C \mid B$ is applicable to a state $H_1' \wedge H_2' \wedge G$ if $CT \models G_B \rightarrow \exists \bar{x}((H_1 = H_1' \wedge H_2 = H_2') \wedge C)$, where $\bar{x}$ are the variables in H_1 and H_2, and G_B is a conjunction of the built-in constraints in G. The state transition is defined as follows:

$$H_1' \wedge H_2' \wedge G \mapsto H_1' \wedge G \wedge B \wedge C \wedge (H_1 = H_1' \wedge H_2 = H_2')$$

3 Operational Equivalence

In Section 4, we gave a quasi formal description of the mapping from $\mathcal{I}$ to CHR programs. We now turn to a more formal presentation of the mapping, ending this section with a proof of equivalence between $\mathcal{I}$ and corresponding CHR programs.

3.1 The Language $\mathcal{I}$

To simplify the exposition, we impose two simple restrictions on $\mathcal{I}$ programs. It should be clear that the following restrictions are only syntactic; the expressive power of $\mathcal{I}$ is preserved.

1. *The identifier on the left-hand side of an assignment statement does not occur on the right-hand side and is not declared in the same statement.*
   This can easily be enforced by the careful use of temporary variables and the separation of declaration from initialization.
2. *No array variables are used.*
   Assuming that all arrays have fixed sizes, an $\mathcal{I}$ program with an array A indexed from 0 to n may be replaced by $n+1$ variables $A_0, \ldots A_n$. Naturally, several other changes will need to be made. In particular, some loops will need to be unwrapped.

The set of thus restricted $\mathcal{I}$ programs may be defined recursively as follows, where we assume the standard imperative syntax of identifiers and expressions.

Definition 1. The set $\mathcal{I}$ is the smallest set containing all of the following forms.

1. $dt\ x$`;`, where dt is a data type and x an identifier
2. x `=` e`;`, where x is an $\mathcal{I}$ identifier and e is an $\mathcal{I}$ expression
3. `if` e $\{P_1\}$ `else` $\{P_2\}$, where e is a Boolean expression and $P_1, P_2 \in \mathcal{I}$
4. `while` e $\{P\}$, where e is a Boolean expression and $P \in \mathcal{I}$
5. $P_1\ P_2$, where $P_1, P_2 \in \mathcal{I}$.

We can provide standard operational semantics for $\mathcal{I}$ in the spirit of [2]. A store σ is a partial function from $\mathcal{I}$ identifiers to $\mathcal{I}$ values. We denote by Σ the set of all possible $\mathcal{I}$ stores. Usually, the semantics is given by a transition system on the set Γ of program configurations, where $\Gamma \subseteq (\mathcal{I} \times \Sigma) \cup \Sigma$. A configuration γ is *terminal* if $\gamma \in \Sigma$. Given that $\mathcal{I}$ programs are deterministic, every terminating program P and initial store σ_i have a unique terminal configuration $[P](\sigma)$. To define the semantics of $\mathcal{I}$, it thus suffices to define the function $[P]$. A recursive definition (on the structure of $\mathcal{I}$ programs) of $[P]$ is given in Figure 1.[1]

In what follows, x is an $\mathcal{I}$ identifier, e is an $\mathcal{I}$ expression, and $\sigma^{x \mapsto v}$ is identical to σ except that $\sigma(x) = v$.

1. $[dt\ x\texttt{;}](\sigma) = \sigma^{x \mapsto Default(dt)}$
   where $Default(dt)$ is the default value for the $\mathcal{I}$ data type dt.
2. $[x\texttt{=}e\texttt{;}](\sigma) = \sigma^{x \mapsto [\![e]\!]^\sigma}$
   where $[\![e]\!]^\sigma$ is the value of the $\mathcal{I}$ expression e with respect to the store σ.
3. $[\texttt{if}\ e\ \{P_1\}\ \texttt{else}\ \{P_2\}](\sigma) = [P](\sigma)$
   where $P = P_1$ if $[\![e]\!]^\sigma$ is true and $P = P_2$ otherwise.
4. $[\texttt{while}\ e\ \{P\}](\sigma) = \gamma$
   where $\gamma = [\texttt{while}\ e\ \{P\}]([P](\sigma))$ if $[\![e]\!]^\sigma$ is true, and $\gamma = \sigma$ otherwise.
5. $[P_1 P_2](\sigma) = [P_2]([P_1](\sigma))$

Fig. 1. Operational semantics of $\mathcal{I}$.

[1] Note that $[P](\sigma)$ is undefined for nonterminating configurations.

3.2 The CHR Fragment

In Section 3.3, we present a mapping from $\mathcal{I}$ to CHR. Naturally, the mapping is not onto, and the image thereof is comprised of CHR programs with only two constraint symbols: `var/2` and `state/2`. The constraint `var` was satisfactorily discussed in Section 4. In this section, we examine the constraint `state/2` and some features of CHR programs employing it. For the purpose of the formal construction, we take `state` to be a binary constraint.
A constraint `state`(b, n) intuitively indicates that the current CHR state corresponds to the state of the $\mathcal{I}$ program following the execution of a statement uniquely identified by the pair (b, n). As per Definition 4, b is a nonempty string over the alphabet $\Sigma_{012} = \{\texttt{0},\ \texttt{1},\ \texttt{2}\}$ starting with a `0`, and n is a non-empty string over the alphabet Σ_D composed of the set of decimal digits and the separator `#`. According to Definition 1, an $\mathcal{I}$ program is a sequence of statements. Each of these statements corresponds to a pair (b, n), where $b = \texttt{0}$ and n is a nonempty string that does not contain the `#` (i.e., a numeral); the number represented by n indicates the order of the statement in the $\mathcal{I}$ program. The special pair (`0`, `0`) corresponds to the state before any statement has been executed. If the statement corresponding to a pair (b, n) is a `while` loop, then a statement in the body of the loop will correspond to the pair $(b, n\texttt{\#}m)$, where m is a numeral denoting the order of the statement within the body of the loop. Similarly, if an `if-then-else` statement corresponds to the pair (b, n), then a statement within the `then` block corresponds to the pair (b`1`, n`#`m). A statement within the `else` block corresponds to the pair (b`2`, n`#`m). In both cases, m is a numeral denoting the order of the statement within the block.
In order to facilitate the definition of the transformation from $\mathcal{I}$ to CHR, we need some terminology to succinctly talk about CHR programs with `state` constraints. We start with two properties of these constraints.
In the sequel, if m and n are numerals, then m_{+n} is the numeral denoting the number $[\![m]\!] + [\![n]\!]$, where $[\![x]\!]$ is the number denoted by the numeral x.

Definition 2. Let P be a CHR program and let $s =$ `state`(b, n) be a constraint in P.

1. s is *terminal* if there is a rule $r = H_1 \setminus s, H_2$ `<=>` C `|` B in P, such that no `state` constraints appear in B. Such a rule r is a *terminal rule*.
2. s is *maximal* if $n = uv$, where v is the longest numeral suffix of n, and for every other constraint `state`$(b', u'v')$ in P, with v' the longest numeral suffix of $u'v'$, b is a substring of b' and either u is a proper substring of u' or $u = u'$ and $[\![v]\!] > [\![v']\!]$.

It is easy to show that if a CHR program has a maximal constraint, then it is unique.

Definition 3. In what follows $P, P1$ and $P2$ are CHR programs, $b \in \Sigma_{012}^{+}$, and $n \in \Sigma_D^{+}$.

1. The *n-translation of* P is the CHR program P_{+n} which is identical to P with every constraint `state`$(b', n'\texttt{\#}m)$ replaced by a constraint `state`$(b', n'\texttt{\#}m_{+n})$.

2. The (b,n)-*nesting of* P is the CHR program $(b,n)\triangleright P$ which is identical to P with every constraint $\texttt{state(0}b',n'\texttt{)}$ replaced by a constraint $\texttt{state(}bb',n\texttt{\#}n'\texttt{)}$.
3. The (b,n)-*termination of* P is the CHR program $(b,n)\triangledown P$ which is identical to P with every terminal rule

$$H_1 \ \backslash \ \texttt{state(}b',n'\texttt{)}, H_2 \texttt{<=>} \ C \,|\, B$$

replaced by the rule

$$H_1 \ \backslash \ \texttt{state(}b',n'\texttt{)}, H_2 \texttt{<=>} \ C \,|\, B, \texttt{state(}b,n\texttt{)}$$

4. Let $\texttt{state(0,}n'\texttt{)}$ be a maximal constraint in P_1. The *concatenation* of P_1 and P_2 is the CHR program

$$P1 \circ P2 = (0,n)\triangledown P1 \cup P2_{+n}$$

where $n = n'_{+1}$.

3.3 The $\mathcal{I}$-CHR Transformation

We can now give the mapping from $\mathcal{I}$ to CHR programs a more formal guise, defining it as a system $\mathcal{T}$ of functions from syntactic $\mathcal{I}$ constructs to syntactic CHR constructs.

Definition 4. An $\mathcal{I}$-CHR transformation is a quadruple $\mathcal{T} = \langle \mathcal{N}, \mathcal{V}, \mathcal{E}, \mathcal{F} \rangle$, where

- $\mathcal{N}$ is an injection from the set of $\mathcal{I}$ identifiers into the set of CHR constants.
- $\mathcal{V}$ is an injection from the set of $\mathcal{I}$ identifiers into the set of CHR variables.
- $\mathcal{E}$ is an injection from the set of $\mathcal{I}$ expressions into the set of CHR expressions, such that $\mathcal{E}(e)$ is similar to e with every identifier x replaced by $\mathcal{V}(x)$, every constant replaced by the equivalent CHR constant, and every operator replaced by the equivalent CHR operator.[2]
- $\mathcal{F} : \mathcal{I} \longrightarrow$ CHR is an injection defined recursively as shown in Figure 2.

The following proposition states some syntactic properties of CHR programs resulting from the above transformation.

Proposition 1. *In what follows P is an $\mathcal{I}$ program and $(b,n) \in \Sigma^+_{012} \times \Sigma^+_D$.*

1. *Every rule in $\mathcal{F}(P)$ has exactly one* `state` *constraint in the head and at most one different* `state` *constraint in the body.*
2. *Every* `state` *constraint occurring in $\mathcal{F}(P)$ occurs in the head of at least one rule.*
3. *$\mathcal{F}(P)$ has a unique maximal constraint.*

[2] Note the implicit, yet crucial, assumption here. We are assuming that there are constant- and operator- bijections between $\mathcal{I}$ and CHR.

In what follows, x is an $\mathcal{I}$ identifier, e is an $\mathcal{I}$ expression, and V is a (possibly empty) conjunction of CHR constraints of the form var($\mathcal{N}(y), \mathcal{V}(y)$), one for each identifier y occurring in e.

1. $\mathcal{F}(dt\ x\texttt{;}) = \{$`state(0,0) <=> var(`$\mathcal{N}(x)$`,` $Default(dt)$`)`$\}$
   where $Default(dt)$ is the default value for the $\mathcal{I}$ data type dt.

2. $\mathcal{F}(x\texttt{=}e\texttt{;}) = \{V$ `\ state(0,0), var(`$\mathcal{N}(x)$`,_) <=>` $\mathcal{V}(x)$`=`$\mathcal{E}(e)$`, var(`$\mathcal{N}(x)$`,`$\mathcal{V}(x)$`)`$\}$

3. $\mathcal{F}($`if` e $\{P_1\}$ `else` $\{P_2\}) =$ `(01,0)` $\triangleright \mathcal{F}(P_1) \cup$ `(02,0)` $\triangleright \mathcal{F}(P_2) \cup S$

   where $S = \{V$ `\ state(0,0) <=>` $\mathcal{E}(e) \mid$ `state(01,0#0)`,
   V `\ state(0,0) <=> \+`$\mathcal{E}(e) \mid$ `state(02,0#0)`$\}$

4. $\mathcal{F}($`while` e $\{P\}) =$ `(0,0)`$\triangledown($`(0,0)` $\triangleright \mathcal{F}(P)) \cup S$

   where $S = \{V$ `\ state(0,0) <=>` $\mathcal{E}(e) \mid$ `state(0,0#0)`,
   V `\ state(0,0) <=> \+`$\mathcal{E}(e)$ `| true`$\}$

5. $\mathcal{F}(P_1 P_2) = \mathcal{F}(P_1) \circ \mathcal{F}(P_2)$

Fig. 2. Definition of the function $\mathcal{F}$ from $\mathcal{I}$ to CHR programs

Note that, had the last statement of the proposition been false, case 5 in Figure 2 would not have made sense. The following important result follows from Definition 4 and Proposition 1.

Theorem 1. *Let P be an $\mathcal{I}$ program. If G is a state containing a single* `state` *constraint that occurs in $\mathcal{F}(P)$, then exactly one rule in $\mathcal{F}(P)$ is applicable to G.*

Corollary 1. *If P is an $\mathcal{I}$ program, then $\mathcal{F}(P)$ is confluent.*

Corollary 2. *If P is an $\mathcal{I}$ program and* `state(0,0)` $\mapsto^*_{\mathcal{F}(P)} G$*, then G contains at most one* `state` *constraint.*

Given Corollary 1, we will henceforth denote the unique final state of $\mathcal{F}(P)$ when started in state G by $[\mathcal{F}(P)](G)$. Note that, given Proposition 1, $[\mathcal{F}(P)](G)$ contains no `state` constraints.

Proposition 2. *In what follows P is an $\mathcal{I}$ program, $(b, n) \in \Sigma^+_{012} \times \Sigma^+_D$, and G is a state containing no* `state` *constraints.*

1. $[\mathcal{F}(P)_{+n}](G \wedge$ `state(0,`n`)`$) = [\mathcal{F}(P)](G \wedge$ `state(0,0)`$)$*, for any numeral n.*
2. $[(b, n) \triangleright \mathcal{F}(P)](G \wedge$ `state(`b, n`#0)`$) = [\mathcal{F}(P)](G \wedge$ `state(0,0)`$)$.
3. $[(b, n)\triangledown\mathcal{F}(P)](G \wedge s) = [\mathcal{F}(P)](G \wedge s) \wedge$ `state(`b, n`)`*, where s is a* `state` *constraint that occurs in $\mathcal{F}(P)$.*

Intuitively, P is equivalent to $\mathcal{F}(P)$ if they have the same effect; that is, if they map equivalent states to equivalent states.

Definition 5. Let $\mathcal{T} = \langle \mathcal{N}, \mathcal{V}, \mathcal{E}, \mathcal{F} \rangle$ be an $\mathcal{I}$-CHR transformation.

1. An $\mathcal{I}$ store σ is equivalent to a CHR state G, denoted $\sigma \equiv G$, whenever $\sigma(x) = v$ if and only if $G = G' \wedge \texttt{var}(\mathcal{N}(x), v)$, where G' is a state that contains no `state` constraints.
2. An $\mathcal{I}$ configuration γ is equivalent to a CHR state G, denoted $\gamma \equiv G$, if either $\gamma = \langle P, \sigma \rangle$ and $G = G' \wedge \texttt{state(0,0)}$ where $\sigma \equiv G'$, or $\gamma = \sigma \equiv G$.
3. An $\mathcal{I}$ program P_1 is equivalent to a CHR program P_2, denoted $P_1 \equiv P_2$, if for every σ and G where $\langle P_1, \sigma \rangle \equiv G$, $[P_1](\sigma) \equiv [P_2](G)$.

Theorem 2. *For every $\mathcal{I}$ program P and every $\mathcal{I}$-CHR transformation $\mathcal{T} = \langle \mathcal{N}, \mathcal{V}, \mathcal{E}, \mathcal{F} \rangle$, $P \equiv \mathcal{F}(P)$.*

Proof. See the appendix.

4 Methodology for the Conversion of Imperative Algorithms to CHR

In this section, we will informally discuss the methodology to convert an algorithm written in a mini imperative programming language, called $\mathcal{I}$, to an equivalent CHR program.
The basic features of the language $\mathcal{I}$ are:

- Variable declaration and assignment
- Alternation using the `if-then-else` commands
- Iteration using the `while-do` command
- Fixed-size arrays

In the following, we present the implementation of each of these features of imperative programming with the intent of implementing an equivalent program in CHR.

4.1 Variable Declaration

In order to create a storage location for a variable, whenever one is declared, a constraint is added to the constraint store and is given the initial value of this variable as a parameter.
The fragment of code

```
int x = 0;
int y = 7;
```

will be transformed into the following CHR rules:

```
r1 @ state(0, 0) <=> var(x,0), state(0, 1).
r2 @ state(0, 1) <=> var(y,7).
```

The constraint `var/2` is used to store the value of the variables. The head of rule `r1` describes the start of the execution of the program by using a constraint `state/1`. Rule `r1` replaces the first `state` constraint by a `var/2` constraint and a new `state` constraint that triggers the execution of the second rule `r2`.
In general, a *variable declaration* in an imperative programming language can be expressed in CHR using a simplification rule of the form:

$$C_{current} \texttt{<=>} V, C_{next}$$

where $C_{current}$ and C_{next} are each a constraint `state/1` with a constant unique parameter. V is a constraint used for the purpose of storing the value of the variable being declared. A constraint V is of the form `var(`*variable,value*`)`.

4.2 Variable Assignment

Assigning a value to an already declared variable in CHR is quite similar to the declaration of the variable. However, instead of adding a constraint with the initial value of the variable, we replace the already existing constraint resulting from the last assignment of a value to the variable with a new `var` constraint with the new assignment.
The fragment of code

```
int x = 0;          // asg1
int y = x + 3;      // asg2
```

will be transformed into the following CHR rules:

```
asg1 @ state(0, 0) <=> var(x,0), state(0, 1).
asg2 @ var(x,V) \ state(0, 1) <=> Y = V + 3, var(y,Y).
```

Rule `asg1` performs a variable declaration with an initial value of `0`. Rule `asg2` uses the value of `x` to compute the value of `y` keeping the same information about `x` in the constraint store.
A *variable assignment* in an imperative programming language can be expressed in CHR using a simpagation rule of the form:

$$V \setminus C_{current}, V_{old} \texttt{<=>} C, V_{new}, C_{next}$$

where V is a conjunction of `var` constraints needed to calculate the new value to be assigned. $C_{current}$ and C_{next} are the same as in the variable declaration rule. V_{old} is the constraint with the old value of the variable which is being assigned a new value, and V_{new} is the same constraint but passed the new value being assigned. C is a conjunction of built-in constraints calculating the new value which is to be assigned. In case the new value being assigned does not depend on values of other variables, both V and C are discarded from the rule and it becomes a simplification rule.

4.3 Alternation

For the fragment of code

```
int a = 10;    // declaration
if(a % 2 == 0)
    a = a * 2; // statement 1
else
    a = a / 2; // statement 2
```

the statements `declaration`, `statement 1`, and `statement 2` are transformed into the following CHR rules:

```
declaration @ state(0, 0) <=> var(a,10), state(0, 1).
statement1 @ state(01, 1#0), var(a,A) <=> NewA = A * 2,  var(a,NewA).
statement2 @ state(02, 1#0), var(a,A) <=> NewA = A // 2,  var(a,NewA).
```

To allow the CHR program to choose whether to execute `statement1` or `statement2` after the `declaration`, we add two rules that are responsible for this choice.

```
goto1 @ var(a,A) \ state(0, 1) <=>
               Tmp = A mod 2, Tmp = 0 |  state(01, 1#0).
goto2 @ var(a,A) \ state(0, 1) <=>
               Tmp = A mod 2, \+(Tmp = 0) |  state(02, 1#0).
```

Alternation in imperative programming, achieved using `if-then-else` expressions can be expressed in CHR using two simpagation rules of the form:

$$V \setminus C_{current} \texttt{<=>} C \mid C_{ifbranch}$$
$$V \setminus C_{current} \texttt{<=>} \neg C \mid C_{elsebranch}$$

where V is a conjunction of `var` constraints needed to evaluate the condition of the `if-then-else` expression. $C_{current}$ is the `state` constraint holding the current state, the state that an `if-then-else` expression is to be executed. $C_{ifbranch}$ is a `state` constraint indicating that the next state is the beginning of the body of the `if` block. $C_{elsebranch}$ is a `state` constraint indicating that the next state is the beginning of the body of the `else` block. C is a guard that evaluates the condition of the `if` statement and $\neg C$ is a guard that evaluates to the negation of C.

4.4 Iteration

Consider the following code fragment

```
int a = 0;     // declaration
while(a < 10)
    a = a + 1; // while block
```

The statements `declaration` and `while block` are transformed into the following CHR rules:

```
declaration @ state(0, 0) <=> var(a,0), state(0, 1).
while_block @ state(0, 1#0), var(a,A) <=>
                 NewA = A + 1, var(a,NewA), state(0, 1).
```

To evaluate the `while-do` condition and add a repetition mechanism for the block of `while-do` as long as this condition holds and to terminate the iteration otherwise, we add the following rules:

```
continue @ var(a,A) \ state(0, 1) <=> A < 10 | state(0, 1#0).
terminate @ var(a,A) \ state(0, 1) <=> \+(A < 10) | true.
```

Iteration in imperative programming, achieved using `while-do` expressions, can be expressed in CHR using the following rules:

$$V \setminus C_{startwhile} \texttt{ <=> } C \mid C_{executebody}$$
$$V \setminus C_{startwhile} \texttt{ <=> } \neg C \mid C_{terminatewhile}$$

where V is a conjunction of `var` constraints needed to evaluate the condition of the `while-do` expression. $C_{startwhile}$ is a `state` constraint holding the current state, the state indicating that a `while-do` expression is to be executed. $C_{executebody}$ is a `state` constraint indicating that the next state is the beginning of the body of the `while` block. $C_{terminatewhile}$ is a `state` constraint indicating that the next state is the beginning of the code following the `while-do` expression, i.e. the termination of the `while-do` expression. $C_{endwhile}$ is a `state` constraint indicating that the block of the `while-do` has ended and that the condition of the loop needs to be checked again. C is a guard that evaluates the condition of `while-do` and $\neg C$ is a guard that evaluates to the negation of C.

4.5 Arrays

To simulate arrays in CHR, we represent them using lists and make use of built-in constraints to either access or modify an element of the list. We assume the existence of the predicate `nth0(N, List, Element)` that holds if `Element` is the `N`th value of the list `List`.
Given `nth0/3`, an access to an array element of the form `x = a[3]` is performed in CHR using a rule of the form:

```
arraysR1 @ var(a,A) \ state(B, N), var(x,_) <=>
               nth0(3, A, Element), var(x,Element), state(B, N+1).
```

where `A` is the list containing the values of the array. `arraysR1` is written according to the rule for variable assignment except that `nth0/3` is used to obtain the value of the element to be assigned to `x`.
We also add the following implementation of `replace0/4` to allow for array element assignment:

```
replace0(List, Index, Value, Result):-
   nth0(Index, List, _, Rest), nth0(Index, Result, Value, Rest).
```

replace0/4 makes use of nth0(N, List, Element, Rest), which behaves similarly to nth0/3 except that Rest is all elements in List other than the Nth element. The resulting predicate replace0/4 sets the element at index Index of List to the value Value and gives the list Result as the new list with the modified element.
We then represent an assignment of the form a[3] = x using a CHR rule of the form:

```
arraysR2 @ var_x(X) \ state(B, N), var(a,A) <=>
   replace0(A, 3, X, NewA),  var(a,NewA), state(B, N+1).
```

arraysR2 is written according to the rule for variable assignment except that replace0/4 is used to obtain the new list NewA which is the new status of the variable a.

Example 1. The following imperative code fragment finds the minimum value in an array a of length n and stores it in a variable min:

```
int temp; int min; int i;
min = a[0];
i = 1;
while(i<n){
   temp = a[i];
   if(temp<min){
      min = temp;
   }
   i = i+1;
}
```

Note that there are no declarations for both a and n as they are expected to be given as input to the program.
Using the conversion method represented above, the following CHR rules are generated.

```
min1 @ state(0, 0) <=> var(temp,0), state(0, 1).
min2 @ state(0, 1) <=> var(min,0), state(0, 2).
min3 @ state(0, 2) <=> var(i,0), state(0, 3).
min4 @ var(a,A) \ state(0, 3), var(min,MIN) <=>
            nth0(0,A,Newmin), var(min,Newmin), state(0, 4).
min5 @ state(0, 4), var(i,I) <=> NewI = 1, var(i,NewI), state(0, 5).
min6 @ var(i,I), var(n,N) \ state(0, 5) <=> (I < N) | state(0, 5#0).
min7 @ var(i,I), var(n,N) \ state(0, 5) <=> \+(I < N) | true.
min8 @ var(i,I), var(a,A) \ state(0, 5#0), var(temp,TEMP) <=>
           nth0(I,A,NewTemp), var(temp,NewTemp), state(0, 5#1).
min9 @ var(temp,TEMP), var(min,MIN) \ state(0, 5#1) <=>
           (TEMP < MIN) | state(01, 5#1#0).
min10 @ var(temp,TEMP), var(min,MIN) \ state(0, 5#1) <=>
           \+(TEMP < MIN) | state(0, 5#2).
```

```
min11 @ var(temp,TEMP) \ state(01, 5#1#0), var(min,MIN) <=>
           NewMIN = TEMP, var(min,NewMIN), state(0, 5#2).
min12 @ state(0, 5#2), var(i,I) <=> NewI = I + 1, var(i,NewI), state(0, 5).
```

To run the CHR program, the following goal is used to pass the necessary constraints and trigger the first rule:

```
var(a, A), length(A, N), var(n, N), state(0, 0).
```

5 Conclusion and Future Work

The context of this paper was a presentation of a conversion methodology to generate rule-based programs from imperative programs. Given a proof of equivalence between both programs, it can be implied that both programs will function alike. The purpose of this generation is to use the rule-based programs in proving properties such as correctness and confluence, which subsequently proves these properties for the imperative programs. We selected CHR as a rule-based language due to the existence of results for proving several properties of programs. There are several implementations of global constraint solvers which are of an imperative nature. An additional use for the implemented conversion methodology could be to automatically generate solvers for these global constraints instead of their manual implementation. The benefit of this conversion is to exploit the flexibility and expressivity of CHR.
An interesting direction for future work is to investigate how the proposed approach can be combined with previous approaches, e.g. [3, 4]. To improve the efficiency of the generated solvers the set of rules should be reduced. The operational equivalence results of CHR programs [1] can be applied to find out the redundant rules. However, in most of the cases, the rules are not redundant but they can be reduced by merging two or more rules in one.

References

1. S. Abdennadher and T. Frühwirth. Operational equivalence of CHR programs and constraints. In *5th International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713, 1999.
2. G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.
3. F. Raiser. Semi-automatic generation of chr solvers for global constraints. In *14th International Conference on Principles and Practice of Constraint Programming*, 2008.
4. I. Sobhi, S. Abdennadher, and H. Betz. Constructing rule-based solvers for intentionally-defined constraints. In *Special Issue on Recent Advances in Constraint Handling Rules*. 2008.

Appendix: Proof of Theorem 2

Let σ be an $\mathcal{I}$ store and let $G = G' \wedge \texttt{state(0,0)}$ be a CHR state, such that $\langle P, \sigma \rangle \equiv G$. Given Definition 5, it suffices to show that $[P](\sigma) \equiv [\mathcal{F}(P)](G)$. We shall prove this result by structural induction on the structure of P.

Basis. We have two base cases.

1. $P = dt\ x$;
   Given the semantics of $\mathcal{I}$, $[P](\sigma) = \sigma^{x \mapsto Default(dt)}$. By Definition 4,
   $$[\mathcal{F}(P)](G) = G' \wedge \texttt{var}(\mathcal{N}(x), Default(dt)\}$$
   It follows from Definition 5 that $[P](\sigma) \equiv [\mathcal{F}(P)](G)$.
2. $P = x\ \texttt{=}\ e$;
   From the semantics of $\mathcal{I}$, $[P](\sigma) = \sigma^{x \mapsto [\![e]\!]^\sigma}$. By Definition 4, if $G' = G'' \wedge \texttt{var}(\mathcal{V}(x), \sigma(x))$ then
   $$[\mathcal{F}(P)](G) = G'' \wedge \texttt{var}(\mathcal{N}(x), [\![\mathcal{E}(e)]\!]^G)$$
   Since $\langle P, \sigma \rangle \equiv G$, it follows that, for every identifier y in e, $\texttt{var}(\mathcal{N}(y), \sigma(y))$ is a constraint in G''. Hence, given the conjunction V of constraints in the head of the only rule in $\mathcal{F}(P)$ (case 2 in Figure 2), the variable $\mathcal{V}(y)$ is bound to $\sigma(y)$, for every identifier y in e. Thus, from the definition of $\mathcal{E}$, it follows that $[\![e]\!]^\sigma = [\![\mathcal{E}(e)]\!]^G$. Consequently, $[P](\sigma) \equiv [\mathcal{F}(P)](G)$.

Induction hypothesis. P_1 and P_2 are $\mathcal{I}$ programs with $P_1 \equiv \mathcal{F}(P_1)$ and $P_2 \equiv \mathcal{F}(P_2)$.

Induction step. We have three recursive rules in the definition of P.

1. $P = \texttt{if}\ e\ \{P_1\}\ \texttt{else}\ \{P_2\}$
   Suppose that $[\![e]\!]^\sigma$ is true. In this case, $[P](\sigma) = [P_1](\sigma)$ as per the operational semantics of $\mathcal{I}$. Now, consider the rule
   $$V \ \texttt{\\}\ \texttt{state(0,0) <=>}\ \mathcal{E}(e) \mid \texttt{state(01,0\#0)}$$
   in $\mathcal{F}(P)$ (case 3 in Figure 2). Similar to case 2 in the proof of the basis, $[\![e]\!]^\sigma = [\![\mathcal{E}(e)]\!]^G$. Thus, the above rule is applicable to G. Furthermore, from Theorem 1, the above rule is the only rule applicable to G. Hence, $G \mapsto_{\mathcal{F}(P)} G_1$, where
   $$G_1 = G' \wedge \texttt{state(01,0\#0)}$$
   Since the set of `state` constraints occurring in $\texttt{(01,0)} \triangleright \mathcal{F}(P_1)$ is disjoint from the set of `state` constraints in the rest of $\mathcal{F}(p)$, and since `state(01,0#0)` occurs in $\texttt{(01,0)} \triangleright \mathcal{F}(P_1)$, then
   $$[\mathcal{F}(P)](G) = [\mathcal{F}(P)](G_1) = [\texttt{(01,0)} \triangleright \mathcal{F}(P_1)](G_1) \qquad (1)$$

From Proposition 2 it follows that

$$[\texttt{(01,0)} \triangleright \mathcal{F}(P_1)](G_1) = [\mathcal{F}(P_1)](G' \wedge \texttt{state(0,0)})$$

But $G' \wedge \texttt{state(0,0)} = G$. Therefore, given (1), $[\mathcal{F}(P)](G) = [\mathcal{F}(P_1)](G)$. From the induction hypothesis it follows that

$$[\mathcal{F}(P)](G) \equiv [P_1](\sigma) = [P](\sigma)$$

The proof is similar, *mutatis mutandis*, in case $[\![e]\!]^\sigma$ is false.

2. $P = \texttt{while}\ e\ \{P_1\}$
We prove the equivalence by induction on the number i of iterations of the loop. If $i = 0$, then it must be that $[\![e]\!]^\sigma$ is false. According to the semantics of I, $[P](\sigma) = \sigma$. We can show (in a fashion similar to that of proving case 2 of the basis) that $[\![e]\!]^\sigma = [\![\mathcal{E}(e)]\!]^G$. Thus, the only rule in $\mathcal{F}(P)$ applicable to G is the rule

$$V\ \texttt{\textbackslash\ state(0,0) <=> \textbackslash+}\mathcal{E}(e)\ \texttt{| true}$$

Since this is a terminal rule, then $[\mathcal{F}(P)](G) = G'$. By Definition 5, $G' \equiv \sigma$. Thus, $[P](\sigma) = [\mathcal{F}(P)](G)$.
As an induction hypothesis, suppose that whenever σ is such that $i = k$, $[P](\sigma) \equiv [\mathcal{F}(P)](G)$. Now, let σ be a store, such that $i = k+1$. Clearly, $[\![e]\!]^\sigma$ is true. Thus, $[P](\sigma) = [P]([P_1](\sigma))$, where $[P_1](\sigma)$ is a store for which $i = k$. It could be shown that $[\![\mathcal{E}(e)]\!]^G = [\![e]\!]^\sigma$. Thus, the only rule in $\mathcal{F}(P)$ applicable to G is the rule

$$V\ \texttt{\textbackslash\ state(0,0) <=> }\mathcal{E}(e) \mid \texttt{state(0,0\#0)}$$

Thus, $G \mapsto_{\mathcal{F}(P)} G_1$, where

$$G_1 = G' \wedge \texttt{state(0,0\#0)}$$

Now, $\texttt{state(0,0)}$ is the only $\texttt{state}$ constraint occurring both in $\texttt{(0,0)}\nabla\texttt{((0,0)}\triangleright \mathcal{F}(P_1))$ and the rest of $\mathcal{F}(P)$. Moreover, according to the definition of ∇, $\texttt{state(0,0)}$ occurs only in the bodies of rules in $\texttt{(0,0)}\nabla\texttt{((0,0)} \triangleright \mathcal{F}(P_1))$. Hence,

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P)](G_1) = [\mathcal{F}(P)]([\texttt{(0,0)}\nabla\texttt{((0,0)} \triangleright \mathcal{F}(P_1))](G_1)) \qquad (2)$$

From Proposition 2 it follows that

$$[\texttt{(0,0)}\nabla\texttt{((0,0)} \triangleright \mathcal{F}(P_1))](G_1) = [\mathcal{F}(P_1)](G' \wedge \texttt{state(0,0)}) \wedge \texttt{state(0,0)}$$

But $G' \wedge \texttt{state(0,0)} = G$. Therefore, given (2), it follows that

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)}) \qquad (3)$$

By the induction hypothesis, $[\mathcal{F}(P_1)](G) \equiv [P_1](\sigma)$. Thus, from Definition 5, $[\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)} \equiv \langle P, [P_1](\sigma)\rangle$. Since $[P_1](\sigma)$ is a store for which $i = k$, then $[P](\sigma) = [P]([P_1](\sigma)) = [\mathcal{F}(P)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)})$. Consequently, given (3), $[P](\sigma) \equiv [\mathcal{F}(P)](G)$.

3. $P = P_1\ P_2$
Let $\texttt{state(0,}n'\texttt{)}$ be the unique maximal constraint in $\mathcal{F}(P_1)$. Given Definition 3, $\texttt{state(0,0)}$ occurs only in the head of a rule in $\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)$, where $n = n'_{+1}$. The only constraint occurring both in $\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)$ and $\mathcal{F}(P_2)_{+n}$ is $\texttt{state(0, }n\texttt{)}$. However, it only occurs in the bodies of rules of $\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)$. Hence,

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P)]([\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)](G)) \tag{4}$$

By Proposition 2,

$$[\texttt{(0,}n\texttt{)}\triangledown\mathcal{F}(P_1)](G) = [\mathcal{F}(P_1)](G) \wedge \texttt{state(0,}n\texttt{)}$$

Hence,

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,}n\texttt{)}) \tag{5}$$

Now, the constraint $\texttt{state(0,}n\texttt{)}$ occurs only in the head of rules in $\mathcal{F}(P_2)_{+n}$. In addition, other $\texttt{state}$ constraints in $\mathcal{F}(P_2)_{+n}$ do not occur elsewhere in $\mathcal{F}(P)$. Hence,

$$[\mathcal{F}(P)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,}n\texttt{)}) = [\mathcal{F}(P_2)_{+n}]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,}n\texttt{)})$$

By Proposition 2,

$$[\mathcal{F}(P_2)_{+n}]([\mathcal{F}(P_1)](s) \wedge \texttt{state(0,}n\texttt{)}) = [\mathcal{F}(P_2)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)})$$

From (5) it follows that

$$[\mathcal{F}(P)](G) = [\mathcal{F}(P_2)]([\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)})$$

But, given the induction hypothesis, $[\mathcal{F}(P_1)](G) \equiv [P_1](\sigma)$. Thus, from Definition 5, $[\mathcal{F}(P_1)](G) \wedge \texttt{state(0,0)} \equiv \langle P_2, [P_1](\sigma)\rangle$. It, thus, also follows from the induction hypothesis that

$$[\mathcal{F}(P)](G) \equiv [P_2]([P_1](\sigma))$$

Hence, given the semantics of $\mathcal{I}$,

$$[\mathcal{F}(P)](G) \equiv [P](\sigma)$$

□

Acknowledgments. We would like to thank Abdellatif Olama for preliminary work done in the same field.