Modular CHR with *ask* and *tell*

François Fages, Cleyton Mario de Oliveira Rodrigues, Thierry Martinez
Contraintes Project–Team, INRIA Paris–Rocquencourt, France

1. Why CHRat?
2. A Simple Example.
3. Syntax and Semantics.
4. Translation of CHRat to flat CHR.
5. Examples of Modular CHRat Solvers.

Programming in CHR.

CHR is a language to define constraint-solvers by multiset rewriting rules which are guarded by built-in constraints.
Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. ***37*** *(1998) 95-138*

Example of constraint solver definition.

Let leq(X,Y) token represent the constraint $X \leq Y$.

(1)	leq(X,X) $\Longleftrightarrow$ true.	← simplifications
(2)	leq(X,Y), leq(Y,X) $\Longleftrightarrow$ X = Y.	← simplifications
(3)	leq(X,Y), leq(Y,Z) $\Longrightarrow$ leq(X,Z).	← propagation
(4)	leq(X,Y) \ leq(X,Y) $\Longleftrightarrow$ true.	← simpagation

Solved forms are irreflexive and transitively closed.

Programming in CHR is non-modular.

Non-reusability of CHR Constraint-Solvers in Guards

Once a new CHR constraint-solver is defined, the resulting solver cannot become the built-in constraint solver of another CHR program.

Satisfaction and Entailment

- CHR constraint-solvers define satisfiability checkers.
- Guards have to be entailed to fire the associated rule.

Towards a Modular CHR Language

Entailment Checking

Three approaches:

1. External implementation
   Duck, G.J., Stuckey, P.J., de la Banda, M.G., Holzbaur, C.: Extending arbitrary solvers with constraint handling rules. In: PPDP'03, Uppsala, Sweden, ACM Press (2003) 79-90
2. Automatic entailment checking

$$C \rightarrow D \dashv\vdash C \wedge D \leftrightarrow C$$

   Schrijvers, T., Demoen, B., Duck, G., Stuckey, P., Frühwirth, T.W.: Automatic implication checking for CHR *constraint solvers. Electronic Notes in Theoretical Computer Science* ***147*** *(2006) 93-11*
3. Our approach: a discipline for programming entailment checking in CHR with *ask* and *tell*.

min Solver over leq Solver in CHR?

Let min(X,Y,Z) represent the constraint that Z is the minimum value among X and Y.

```
leq(X,Y) \ min(X,Y,Z) <=> Z=X.
leq(Y,X) \ min(X,Y,Z) <=> Z=Y.
min(X,Y,Z) ==> leq(Z,X), leq(Z,Y).
```

Does not work: min(X,X,Z) will not be rewritten to X=Z because there is no leq(X,X) token in the store.

leq Solver Component in CHRat.

File leq_solver.cat

```
component leq_solver.
export leq/2.
leq(X,X) <=> true.
leq(X,Y), leq(Y,X) <=> X = Y.
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
leq(X,Y) \ leq(X,Y) <=> true.

ask(leq(X,X))<=>entailed(leq(X,X)).
```

```
leq(X,Y)\ask(leq(X,Y)) <=> entailed(leq(X,Y)).
```

min Solver Component in CHRat.

File min_solver.cat

```
component min_solver.
import leq/2 from leq_solver.
export min/3.
min(X,Y,Z) <=> leq(X,Y) | Z=X.
min(X,Y,Z) <=> leq(Y,X) | Z=Y.
min(X,Y,Z) ==> leq(Z,X), leq(Z,Y).

ask(min(X, Y, X)) <=> leq(X, Y) |
                      entailed(min(X, Y, X)).
ask(min(X, Y, Y)) <=> leq(Y, X) |
                      entailed(min(X, Y, Y)).
```

```
min(X,Y,Z)\ask(min(X,Y,Z)) <=> entailed(min(X,Y,Z)).
```

CHRat Syntax.

component *<component-name>.* one per file.

import *<constraint-declarations>* **from** *<component-name>.*
separation is atom-prefix based.

export *<constraint-declarations>.*

$$\text{<rule-name>}\ \texttt{@}\ <\mathcal{H}>\ \backslash\ <\mathcal{H}> \longleftrightarrow <\mathcal{C}>,<\mathcal{T}>\ |\ <\mathcal{B}>.$$

where:

- $\mathcal{C}$: built-in constraints
- $\mathcal{T}$: CHR constraints
- $\mathcal{H} \doteq \mathcal{T} \uplus \text{ask}(\mathcal{T})$
- $\mathcal{B} \doteq \mathcal{C} \uplus \mathcal{T} \uplus \text{entailed}(\mathcal{T})$

Side condition Every variable which appears in a CHR guard must appear in the built-in guard or in the heads of the rule.

CHRat Operational Semantics for Rules. (1/3)

Configurations $\langle \underbrace{F}_{\text{query}}, \underbrace{E}_{\substack{\text{CHR}\\\text{store}}}, \underbrace{D}_{\substack{\text{built-in}\\\text{store}}} \rangle_{\mathcal{V}}$

where $\mathcal{V}$ is the set of free variables of the initial query.

Logical meaning $\exists \vec{y}(\overline{F} \wedge \overline{E} \wedge D)$, where $\vec{y}$ enumerates $\mathrm{fv}\,(F, E, D) \setminus \mathcal{V}$.

CHRat Operational Semantics for Rules. (2/3)

Solve

$$\frac{c \in \mathcal{C}}{\langle \{c\} \uplus F, E, D \rangle_{\mathcal{V}} \mapsto \langle F, E, c \wedge D \rangle_{\mathcal{V}}}$$

Introduce

$$\frac{t \in \mathcal{T}^{\bullet}}{\langle \{t\} \uplus F, E, D \rangle_{\mathcal{V}} \mapsto \langle F, \{t\} \uplus E, D \rangle_{\mathcal{V}}}$$

where $\mathcal{T}^{\bullet} = \mathcal{T} \uplus \mathrm{ask}(\mathcal{T}) \uplus \mathrm{entailed}(\mathcal{T})$.

Trivial Entailment

$$\frac{t \in \mathcal{T}}{\langle F, \{\mathrm{ask}(t), t\} \uplus E, D \rangle_{\mathcal{V}} \mapsto \langle \{\mathrm{entailed}(t)\} \uplus F, \{t\} \uplus E, D \rangle_{\mathcal{V}}}$$

CHRat Operational Semantics for Rules. (3/3)

Ask

$$\frac{(H \ \backslash \ H' \ \Leftrightarrow \ C_b, C_c \mid B.)\,\sigma \in P \qquad D \vdash_{\mathcal{C}} C_b}{\langle F, H \uplus H' \uplus E, D\rangle_{\mathcal{V}} \mapsto \langle \mathrm{ask}(C_c) \uplus F, H \uplus H' \uplus E, D\rangle_{\mathcal{V}}}$$

Fire

$$\frac{(H \ \backslash \ H' \ \Leftrightarrow \ C_b, C_c \mid B.)\,\sigma \in P \qquad D \vdash_{\mathcal{C}} C_b}{\langle F, H \uplus H' \uplus \mathrm{entailed}(C_c) \uplus E, D\rangle_{\mathcal{V}} \mapsto \langle B \uplus F, H \uplus E, D\rangle_{\mathcal{V}}}$$

CHRat Declarative Semantics for Rules.

$$\left(H \ \backslash \ H' \ \Leftrightarrow \ C_b, C_c \mid B.\right)^{\ddagger} \doteq$$
$$\forall \vec{y}(C_b \rightarrow \overline{H} \wedge \overline{H'} \rightarrow \overline{\text{ask}(C_c)})$$
$$\wedge \forall \vec{y}(C_b \rightarrow (\overline{H} \wedge \overline{H'} \wedge \overline{\text{entailed}(C_c)} \leftrightarrow \exists \vec{y}'(\overline{H} \wedge \overline{B})))$$

Theorem

Operational semantics is sound and complete with respect to declarative semantics.
If $\mathcal{D}$ is the declarative semantics of a program P and $S_1 \mapsto S_2$ two successive configurations in an execution of P, then:

$$D \vdash_{\mathcal{C}} S_1 \leftrightarrow S_2$$

Adapted from the soundness and completeness theorem of CHR:
Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. ***37*** *(1998) 95-138*

Translation to flat CHR.

$$[\![H \ \backslash \ H' \ \Leftrightarrow \ C_b, C_c \mid B.]\!]$$

$$\doteq \begin{cases} H, H' \ \Rightarrow \ C_b \mid \text{ask}(C_c). \\ H \ \backslash \ H', \text{entailed}(C_c) \ \Leftrightarrow \ C_b \mid B. \end{cases}$$

Theorem

If:

- $\mathcal{D}$ *is the* CHRat *declarative semantics of a* CHRat *program* P*; and*
- $\mathcal{D}'$ *is the* CHR *declarative semantics of* $[\![P]\!]$.

then:

$$\vdash_{\mathcal{C}} \mathcal{D} \leftrightarrow \mathcal{D}'$$

Example of Translation to flat CHR.

```
min(X,Y,Z)\ask_min(X,Y,Z) ==> entailed_min(X,Y,Z).
```

```
min(X,Y,Z) <=> leq(X,Y) | Z=X.
```

```
min(X,Y,Z) ==> ask_leq(X,Y).
entailed_leq(X,Y), min(X,Y,Z) <=> Z=X.
```

```
min(X,Y,Z) ==> leq(Z,X), leq(Z,Y).
```

```
min(X,Y,Z) ==> leq(Z,X), leq(Z,Y).
```

```
ask(min(X, Y, X)) <=> leq(X, Y) |
                        entailed(min(X, Y, X)).
```

```
ask_min(X,Y,X) ==> ask_leq(X,Y).
entailed_leq(X,Y), ask_min(X,Y,X) <=>
                          entailed_min(X,Y,X).
```

Union-find Component. (1/3)

Satisfiability solver comes from *Schrijvers, T., Frühwirth, T.W.: Analysing the* CHR *implementation of unionfind. In: 19th Workshop on (Constraint) Logic Programming. (2005)*

File union_find_solver.cat

```
component union_find.
export make/1, ≃/2.
make(A) ⟺ root(A, 0).

union(A, B) ⟺ find(A, X), find(B, Y), link(X, Y).

A ⇝ B, find(A, X) ⟺ find(B, X), A ⇝ X.
root(A, _) \ find(A, X) ⟺ X = A.

link(A, A) ⟺ true.
link(A, B), root(A, N), root(B, M) ⟺ N ≥ M |
                B ⇝ A, N1 is max(M+1, N), root(A, N1).
link(B, A), root(A, N), root(B, M) ⟺ N ≥ M |
                B ⇝ A, N1 is max(M+1, N), root(A, N1).
```

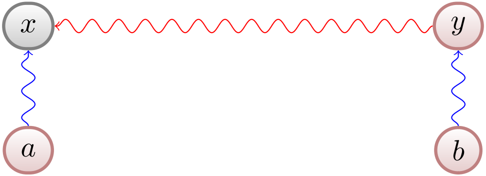
Union-find Component. (2/3)

File union_find_solver.cat

```
A ≃ B ⟹ union(A, B).

ask(A ≃ B) ⟺
                 find(A, X), find(B, Y),
                 check(A, B, X, Y).
root(X) \ check(A, B, X, X) ⟺
                 entailed(A ≃ B).
```

Union-find Component. (3/3)




```
X ⇝ C \ check(A, B, X, Y) ⟺
                 find(A, Z), check(A, B, Z, Y).
Y ⇝ C \ check(A, B, X, Y) ⟺
                 find(B, Z), check(A, B, X, Z).
```

Rational Tree Solver Component.

File rational_tree_solver.cat

```
component rational_tree_solver.
import ≃/2 from union_find_solver.
export fun/3, arg/3, ∼/2.
fun(X0, F0, N0) \ fun(X1, F1, N1) ⟺ X0 ≃ X1 |
            F0 = F1, N0 = N1.
arg(X0, N, Y0) \ arg(X1, N, Y1) ⟺ X0 ≃ X1 |
            Y0 ≃ Y1.

X ∼ Y ⟺ X ≃ Y.
```

Check the paper for the ask-solver!

Conclusion.

Objective.

- Generalization of guards.
- Modular definition of solvers.

Proposed Solution.

- Programming discipline to define satisfiability *and* entailment constraint solvers.

Perspectives.

- Relax the restriction on guard variables.
- Link between declarative semantics of ask and logical implication.
- Modular compilation.