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.
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.


Example of constraint solver definition.

Let \( \text{leq}(X,Y) \) token represent the constraint \( X \leq Y \).

\[\begin{align*}
(1) \quad & \text{leq}(X,X) \iff \text{true.} \\
(2) \quad & \text{leq}(X,Y), \text{leq}(Y,X) \iff X = Y. \\
(3) \quad & \text{leq}(X,Y), \text{leq}(Y,Z) \implies \text{leq}(X,Z). \iff \text{propagation} \\
(4) \quad & \text{leq}(X,Y) \setminus \text{leq}(X,Y) \iff \text{true.} \iff \text{simpagation}
\end{align*}\]

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


2. Automatic entailment checking

   \[ C \rightarrow D \iff C \land D \leftrightarrow C \]


3. Our approach: a discipline for programming entailment checking in CHR with *ask* and *tell*.
Let \( \min(X,Y,Z) \) represent the constraint that \( Z \) is the minimum value among \( X \) and \( Y \).

\[
\begin{align*}
\text{leq}(X,Y) \setminus \min(X,Y,Z) & \iff Z=X. \\
\text{leq}(Y,X) \setminus \min(X,Y,Z) & \iff Z=Y. \\
\min(X,Y,Z) & \implies \text{leq}(Z,X), \text{leq}(Z,Y).
\end{align*}
\]

Does not work: \( \min(X,X,Z) \) will not be rewritten to \( X=Z \) because there is no \( \text{leq}(X,X) \) 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>`.


```latex
<rule-name> @ <H> \setminus <H> \iff <C>, <T> | <B>.
```

where:

- **C**: built-in constraints
- **T**: CHR constraints
- **H = T \sqcup ask(T)**
- **B = C \sqcup T \sqcup \text{entailed}(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.
Configurations $\langle F, E, D \rangle_V$

where $V$ is the set of free variables of the initial query.

Logical meaning $\exists \vec{y}(\overline{F} \land \overline{E} \land D)$, where $\vec{y}$ enumerates $\text{fv}(F, E, D) \setminus V$. 
Solve

\[
    c \in \mathcal{C} \\
    \langle \{c\} \cup F, E, D \rangle_\mathcal{V} \leftrightarrow \langle F, E, c \land D \rangle_\mathcal{V}
\]

Introduce

\[
    t \in \mathcal{T}^* \\
    \langle \{t\} \cup F, E, D \rangle_\mathcal{V} \leftrightarrow \langle F, \{t\} \cup E, D \rangle_\mathcal{V}
\]

where \( \mathcal{T}^* = \mathcal{T} \cup \text{ask}(\mathcal{T}) \cup \text{entailed}(\mathcal{T}) \).

Trivial Entailment

\[
    t \in \mathcal{T} \\
    \langle F, \{\text{ask}(t), t\} \cup E, D \rangle_\mathcal{V} \leftrightarrow \langle \{\text{entailed}(t)\} \cup F, \{t\} \cup E, D \rangle_\mathcal{V}
\]
Ask

\[
(H \setminus H' \iff C_b, C_c \mid B.) \sigma \in P \quad D \vdash_{c} C_b
\]

\[
\langle F, H \cup H' \cup E, D \rangle_{\mathcal{V}} \leftrightarrow \langle \text{ask}(C_c) \cup F, H \cup H' \cup E, D \rangle_{\mathcal{V}}
\]

Fire

\[
(H \setminus H' \iff C_b, C_c \mid B.) \sigma \in P \quad D \vdash_{c} C_b
\]

\[
\langle F, H \cup H' \cup \text{entailed}(C_c) \cup E, D \rangle_{\mathcal{V}} \leftrightarrow \langle B \cup F, H \cup E, D \rangle_{\mathcal{V}}
\]
Theorem

**Operational semantics is sound and complete with respect to declarative semantics.**

If $D$ is the declarative semantics of a program $P$ and $S_1 \mapsto S_2$ two successive configurations in an execution of $P$, then:

$$D \vdash_c S_1 \leftrightarrow S_2$$

Adapted from the soundness and completeness theorem of CHR:

Translation to flat CHR.

\[
\begin{align*}
[H \ \ \ H'] & \iff C_b, C_c \mid B. \\
\end{align*}
\]

\[
\begin{align*}
\vdash H, H' \Rightarrow C_b \mid \text{ask}(C_c). \\
H \ \ \ H', \text{entailed}(C_c) & \iff C_b \mid B.
\end{align*}
\]

Theorem

If:

- \(D\) is the CHRat declarative semantics of a CHRat program \(P\); and
- \(D'\) is the CHR declarative semantics of \(\llbracket P \rrbracket\).

then:

\[\vdash_c D \leftrightarrow D'\]
Example of Translation to flat CHR.

\[ \text{min}(X, Y, Z) \setminus \text{ask}_{-}\text{min}(X, Y, Z) \implies \text{entailed}_{-}\text{min}(X, Y, Z). \]

\[ \text{min}(X, Y, Z) \iff \text{leq}(X, Y) \mid Z=X. \]

\[ \text{min}(X, Y, Z) \implies \text{ask}_{-}\text{leq}(X, Y). \]
\[ \text{entailed}_{-}\text{leq}(X, Y), \text{min}(X, Y, Z) \iff Z=X. \]

\[ \text{min}(X, Y, Z) \implies \text{leq}(Z, X), \text{leq}(Z, Y). \]

\[ \text{ask}(\text{min}(X, Y, X)) \iff \text{leq}(X, Y) \mid \text{entailed}(\text{min}(X, Y, X)). \]

\[ \text{ask}_{-}\text{min}(X, Y, X) \implies \text{ask}_{-}\text{leq}(X, Y). \]
\[ \text{entailed}_{-}\text{leq}(X, Y), \text{ask}_{-}\text{min}(X, Y, X) \iff \text{entailed}_{-}\text{min}(X, Y, X). \]
Union-find Component. (1/3)


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

\[ A \simeq B \implies \text{union}(A, B). \]

\[ \text{ask}(A \simeq B) \iff \text{find}(A, X), \text{find}(B, Y), \]
\[ \text{check}(A, B, X, Y). \]

\[ \text{root}(X) \setminus \text{check}(A, B, X, X) \iff \text{entailed}(A \simeq B). \]
Union-find Component. (3/3)

\[ x \leadsto C \setminus \text{check}(A, B, X, Y) \iff \text{find}(A, Z), \text{check}(A, B, Z, Y). \]

\[ Y \leadsto C \setminus \text{check}(A, B, X, Y) \iff \text{find}(B, Z), \text{check}(A, B, X, Z). \]
Rational Tree Solver Component.

File rational_tree_solver.cat

component rational_tree_solver.

import \equiv/2 from union_find_solver.

export fun/3, arg/3, \equiv/2.

fun(X0, F0, N0) \iff fun(X1, F1, N1) \iff X0 \equiv X1 \land F0 = F1, N0 = N1.

arg(X0, N, Y0) \iff arg(X1, N, Y1) \iff X0 \equiv X1 \land Y0 \equiv Y1.

X \equiv Y \iff X \equiv 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.