Satisfiability Modulo Constraint Handling Rules (Extended Abstract)*

Gregory J. Duck
School of Computing, National University of Singapore
gregory@comp.nus.edu.sg

Abstract

Satisfiability Modulo Constraint Handling Rules (SMCHR) is the integration of the Constraint Handling Rules (CHR) solver programming language into a Satisfiability Modulo Theories (SMT) solver framework. Constraint solvers are implemented in CHR as a set of high-level rules that specify the simplification (rewriting) and constraint propagation behavior. The traditional CHR execution algorithm manipulates a global store representing a flat conjunction of constraints. This paper introduces SMCHR: a tight integration of CHR with a modern Boolean Satisfiability (SAT) solver. Unlike CHR, SMCHR can handle (quantifier-free) formulas with an arbitrary propositional structure. SMCHR is essentially a Satisfiability Modulo Theories (SMT) solver where the theory \( T \) is implemented in CHR.

1 Introduction

Constraint Handling Rules (CHR) [Frühwirth, 1998] is an established [Sneyers et al., 2010] rule-based programming language for the specification and implementation of constraint solvers. CHR has two main types of rules: simplification rules \((H \iff B)\) that rewrites constraints \( H \) to \( B \), and propagation rules \((H \implies B)\) that adds (propagates) constraints \( B \) for every \( H \). Constraint solvers are specified as sets of rules.

Example 1 (Bounds Propagation Solver). A bounds propagation solver propagates constraints of the form \((x \geq l)\) and \((x \leq u)\) for numeric constants \( l \) (lower bound) and \( u \) (upper bound). We can specify bounds propagation through addition via the following rules:

\[
\begin{align*}
\text{plus}(x, y, z) & \land \text{lb}(y, l_y) \land \text{lb}(z, l_z) \implies \text{lb}(x, l_y + l_z) \\
\text{plus}(x, y, z) & \land \text{ub}(y, u_y) \land \text{ub}(z, u_z) \implies \text{ub}(x, u_y + u_z)
\end{align*}
\]

Here \( \text{plus}(x, y, z) \) represents \( x = y + z \), \( \text{lb}(x, l) \) represents \( x \geq l \), and \( \text{ub}(x, u) \) represents \( x \leq u \). Given an initial goal \( \text{plus}(a, b, c) \land \text{lb}(h, 3) \land \text{ub}(b, 10) \land \text{lb}(c, 4) \land \text{ub}(c, 6) \), the rules will propagate \( \text{lb}(a, 7) \land \text{ub}(a, 16) \). We can similarly write rules to propagate bounds in other directions.

The operational semantics of CHR [Frühwirth, 1998] [Duck et al., 2004] [Sneyers et al., 2010] manipulate a global constraint store. The store represents a flat conjunction of constraints. By default CHR does not support goals/stores that are formulae with a rich propositional structure, i.e. containing disjunction, negation, etc. Some CHR systems, such as the K.U.Leuven CHR system [Schrijvers and Demoen, 2004], rely on the host language to provide such functionality. For example, using Prolog’s backtracking search to implement disjunction.

In this paper we use a different approach: we extend CHR with a modern Boolean Satisfiability (SAT) solver to form Satisfiability Modulo Constraint Handling Rules (SMCHR). The idea is to specify constraint solvers using CHR in the usual way, such as the rules in Example 1. SMCHR goals are then quantifier-free formulae of CHR constraints over any arbitrary propositional structure.

Example 2 (SMCHR Goal). For example, the following SMCHR goal encodes the classic n-queens problem for the instance \( n = 2 \).

\[
(Q_1 = 1 \lor Q_1 = 2) \land (Q_2 = 1 \lor Q_2 = 2) \land \\
\neg(Q_1 = Q_2) \land \neg(Q_1 = Q_2 + 1) \land \neg(Q_2 = Q_1 + 1)
\]

This goal can be evaluated using an extended version of the bounds propagation solver from Example 1. For \( n = 2 \) the goal is unsatisfiable.

SMCHR is essentially an extensible Satisfiability Modulo Theory (SMT) solver [Moura and Björner, 2011] where the theory \( T \) solver is implemented in CHR. CHR is a well established [Sneyers et al., 2010] solver specification/implementation language, and is therefore a natural choice for implementing theory solvers.

This paper is organized as follows: Section 2 introduces the CHR language, Section 3 introduces the extended SMCHR language, Section 4 presents the SMCHR execution algorithm DPLL(CHR), Section 5 presents an experimental evaluation, and in Section 6 we conclude.

2 Constraint Handling Rules

This section presents an informal overview of Constraint Handling Rules (CHR). For a formal treatment,
CHR is a rule-based programming language with three types of rules:

\[ H \leftrightarrow B \quad \text{(simplification)} \]
\[ H \Rightarrow B \quad \text{(propagation)} \]
\[ H_1 \setminus H_2 \leftrightarrow B \quad \text{(simplification)} \]

where the head \( H \), \( H_1 \), \( H_2 \), and body \( B \) are conjunctions of constraints. CHR solvers apply rules to a set (representing a conjunction) of constraints \( S \) known as the constraint store. The store may contain both CHR constraints (as defined by the rules) and/or built-in constraints such as equality \( x = y \), etc. Given a store \( S \), subsets \( H' \subseteq E \subseteq S \) where \( E \) are equality constraints, and a matching substitution \( \theta \) such that \( \mathcal{E} \theta H = H' \), then a simplification rule \( (H \Rightarrow B) \) rewrites \( H' \) to \( (\theta, B) \), i.e. \( S := (S \setminus H') \cup (\theta, B) \). Likewise, a propagation rule \( (H \Rightarrow B) \) adds \( (\theta, B) \) whilst retaining \( H' \), i.e. \( S := S \cup (\theta, B) \). Finally a simplification rule \( (H_1 \setminus H_2 \Rightarrow B) \) is a hybrid between a simplification and propagation rule, where only the constraints \( H_2 \subseteq S \) matching \( H_2 \) are rewritten, i.e. \( S := (S \setminus H_2') \cup (\theta, B) \). The body \( B \) may also contain built-in constraints. The CHR execution algorithm repeatedly applies rules to the store until a fixed-point is reached or failure occurs.

**Example 3** (Less-than-or-equal-to Solver). The following is a simple “less-than-or-equal-to” solver implemented in CHR:

\[ \text{leq}(X, X) \leftrightarrow \text{true} \quad (1) \]
\[ \text{leq}(X, Y) \land \text{leq}(Y, X) \leftrightarrow X = Y \quad (2) \]
\[ \text{leq}(X, Y) \land \text{leq}(Y, Z) \leftrightarrow \text{leq}(X, Z) \quad (3) \]

This solver defines a leq constraint with three rules. Rule (1) simplifies any constraint of the form \( \text{leq}(X, X) \) to the true constraint. Rule (2) simplifies constraints \( \text{leq}(X, Y) \land \text{leq}(Y, X) \) to \( X = Y \). Finally, propagation rule (3) adds a constraint \( \text{leq}(X, Z) \) for every pair \( \text{leq}(X, Y) \land \text{leq}(Y, Z) \).

Consider the initial store (a.k.a. the goal) \( G = \{ \text{leq}(A, B) \land \text{leq}(B, C) \land \text{leq}(A, C) \} \), then one possible execution sequence (a.k.a. derivation) is as follows:

\[ \{ \text{leq}(A, B), \text{leq}(B, C), \text{leq}(A, C) \} \]  \quad \text{Apply (3)}
\[ \rightarrow \{ \text{leq}(A, B), \text{leq}(B, C), \text{leq}(A, C), \text{leq}(A, C) \} \]  \quad \text{Apply (2)}
\[ \rightarrow \{ \text{leq}(A, B), \text{leq}(B, C), A = C \} \]  \quad \text{Apply (2)}
\[ \rightarrow \{ A = B, A = C \} \]

Here the underlined constraints indicate where the rule is applied. Execution proceeds by first applying the propagation rule (3) which adds the constraint \( \text{leq}(A, C) \). Next, the simplification rule (2) is applied twice, replacing the leq constraints with equalities. In the final store, no more rules are applicable, so execution stops. In general there may be more than one possible derivation for a given goal.

The logical semantics (or logical interpretation) \( \llbracket R \rrbracket \) of a given rule \( R \) is defined as follows:

\[ \llbracket H \leftrightarrow B \rrbracket = \forall(H \leftrightarrow B) \]
\[ \llbracket H \Rightarrow B \rrbracket = \forall(H \rightarrow B) \]
\[ \llbracket H_1 \setminus H_2 \leftrightarrow B \rrbracket = \forall(H_1 \land H_2 \leftrightarrow H_1 \land B) \]

where \( \forall F \) represents the universal closure of \( F \). Here and throughout this paper we assume \( \text{vars}(B) \subseteq \text{vars}(H) \).

**Example 4** (Logical Semantics). The logical interpretation of each rule from Example 3 is a corresponding partial order axiom, namely: rule (1) is reflexivity \( \forall x : x \leq x \), rule (2) is antisymmetry \( \forall x, y : x \leq y \land y \leq x \leftrightarrow x = y \), and rule (3) is transitivity \( \forall x, y, z : x \leq y \land y \leq z \rightarrow x \leq z \).

Note the close correspondence between the syntax of CHR and the logical interpretation.

## 3 Satisfiability Modulo CHR

**Satisfiability Modulo Constraint Handling Rules (SMCHR)** differs from CHR in several ways. This section outlines the differences.

The SMCHR language is an extension of CHR. Unlike CHR, SMCHR allows negation in rules.

**Example 5** (Negation in SMCHR). If we assume that \( \text{leq} \) is a total order relation, then we can extend Example 3 with the following rules defining the negation of \( \text{leq} \):

\[ \lnot \text{leq}(X, Y) \land \lnot \text{leq}(Y, X) \Rightarrow \text{false} \quad (4) \]
\[ \lnot \text{leq}(X, Y) \land \lnot \text{leq}(Y, Z) \Rightarrow \lnot \text{leq}(X, Z) \quad (5) \]

Operationally, these rules match negated constraints that explicitly appear in the store, e.g:

\[ \{ \lnot \text{leq}(A, B), \lnot \text{leq}(B, C), \lnot \text{leq}(A, C) \} \]

**Apply (5)**

\[ \lnot \{ \lnot \text{leq}(A, B), \lnot \text{leq}(B, C), \lnot \text{leq}(A, C) \} \]

The logical semantics of CHR is also extended to allow for negation. The logical interpretation for the above rules is:

\[ \forall x, y : \lnot(x \leq y) \land \lnot(y \leq x) \rightarrow \text{false} \]
\[ \forall x, y, z : \lnot(x \leq y) \land \lnot(y \leq z) \rightarrow \lnot(x \leq z) \]

Other key differences between CHR and SMCHR include:

- **Range-Restricted**: We assume all SMCHR rules are range restricted, i.e. for rule head \( H \) and body \( B \) we have that \( \text{vars}(B) \subseteq \text{vars}(H) \). CHR does not have such a restriction.

- **Set-Semantics**: CHR uses a multi-set semantics by default, meaning that more than one copy of a constraint may appear in the store at once. SMCHR assumes set-semantics that assumes at most one copy. This is equivalent to assuming the following rules are “built-in” for each constraint symbol \( c \):

\[ c(x) \setminus c(x) \leftrightarrow \text{true} \text{ and } c(x) \land \lnot c(x) \rightarrow \text{false} \]

Set-Semantics ensures that each constraint \( c(x) \) can be associated with exactly one propositional variable \( b \). This simplifies the overall design of the SMCHR system.

Goals in CHR are flat conjunctions of constraints. In contrast, SMCHR allows for goals that are any (quantifier-free) formulae with an arbitrary propositional structure, such as that shown in Example 2. Range-Restricted CHR programs ensure that no new (existentially) quantified variables are introduced by rule application.

For a given goal \( G \) and a CHR program \( P \), SMCHR generates one of two possible answers: UNSAT or UNKNOWN.
Otherwise, the algorithm has constructed a valuation setting \( l \) explored (represented by \( \emptyset \)) and the search resumes. If the entire search space has been explored, this behavior mirrors CHR: if \( G \rightarrow^* \) \false, then \( G \) is unsatisfiable. Otherwise if \( G \rightarrow^* S \neq \false \), then it is unknown if \( G \) is (un)satisfiable.

### 4 DPLL(CHR): Execution Algorithm

The SMCHR execution algorithm is based on a variant of the Davis-Putnam-Logemann-Loveland (DPLL) [Davis et al., 1962] decision procedure for propositional formulae combined with CHR solving, i.e. DPLL(CHR).

The first step is to translate the goal \( G \) into normal form \( B \land D \) such that:

1. \( B \) is a pure propositional formula in CNF;
2. \( D \) is a conjunction of equivalences of the form \( b \leftrightarrow c(x) \) where \( b \) is a propositional variable and \( c(x) \) is a constraint; and
3. For all valuations (functions mapping variables to values) \( s \) there exists a valuation \( s' \) such that

\[
\square P \models s(G) \text{ iff } \square P \models s'(B \land D)
\]

and \( s(v) = s'(v) \) for all \( v \in \text{vars}(G) \).

The last condition ensures that \( B \land D \) is equisatisfiable to \( G \) and the solutions of \( G \) and \( B \land D \) correspond. There may be many possible normalizations for a given goal \( G \), and the exact normalization algorithm is left to the implementation.

The propositional component \( B \) is solved with a Boolean Satisfiability (SAT) solver using a variant of DPLL algorithm. Let \( \text{Clauses} \) be the set of clauses in \( B \). We assume each clause is a set of the form \( \{l_1, \ldots, l_n\} \) (representing the disjunction \( l_1 \lor \ldots \lor l_n \), where each \( l_i \) is a literal \( b \) or \( \neg b \) for some propositional variable \( b \in \text{vars}(B) \). The pseudo-code for the DPLL algorithm is as follows:

```plaintext
function dpll(Clauses) =
    while \exists b \in \text{vars}(Clauses) : \text{unset}(b)
        l := selectLiteral(Clauses)
        Clauses := setLiteral(Clauses, l)
        Clauses := unitPropagate(Clauses)
        if \emptyset \in Clauses
            Clauses := backtrack(Clauses)
            if Level = 0 return UNSAT
        return UNKNOWN
```

The DPLL algorithm works by periodically selecting literals \( l \) (via selectLiteral()) and setting \( l \) to true (via setLiteral()). Next unit propagation (via unitPropagate()) propagates the change through Clauses by eliminating unit clauses \( \{l\} \) and setting \( l \). If this process generates the empty clauses \( \emptyset \), then the algorithm backtracks (via backtrack()) to a previous state and the search resumes. If the entire search space has been explored (represented by Level = 0), then UNSAT is returned. Otherwise, the algorithm has constructed a valuation \( s \) over the propositional variables \( \text{vars}(B) \) that satisfies \( B \), and the answer UNKNOWN is returned. Our pseudo-code is a simplification. An actual SMCHR implementation will typically use a modern SAT solver design [Een and Sörensson, 2003] with conflict-driven search, no-good learning, backjumping, etc.

The CHR component of the SMCHR system maintains a global \textit{Store} of constraints (as shown in Example 3). Whenever the SAT solvers sets a literal \( l \in \{b, \neg b\} \) for propositional variable \( b \in \text{vars}(B) \), a chrPropagate() routine is invoked, which is defined as follows:

```plaintext
function chrPropagate(l, Clauses) =
    if \( D \equiv (b \leftrightarrow c(x) \land \ldots) \)
        Store := Store \cup \{(l \equiv b? c(x) : \neg c(x)\}
    let ChrClauses := chrMatch(Rules, Store)
    Clauses := Clauses \cup ChrClauses
    return Clauses
```

If \( l \) (via \( b \)) corresponds to a constraint \( c(x) \), then the \text{Store} is updated to include \( c(x) \) if \( l \equiv b \) or \( \neg c(x) \) if \( l \equiv \neg b \). Next the chrMatch routine attempts to match (and apply) a CHR rule. We shall use the notation \( c(x) \) to represent a constraint literal \( c(\bar{x}) \) or \( \neg c(\bar{x}) \). The chrMatch routine searches for a (renamed apart) rule \((H_1 \land H_2 \iff B) \in \text{Rules} \) and sets of constraints \( C_1, C_2, E \subseteq \text{Store} \) such that \( E \) is a minimal set of equality constraints, and for \( C_1 \cup C_2 = \{c_1(z_1), \ldots, c_n(z_n)\} \) and \( H_1 \cup H_2 = \{\hat{c}_1(y_1), \ldots, \hat{c}_n(y_n)\} \) there exists a matching substitution \( \theta \) such that \( E \rightarrow \theta.\hat{c}_i(y_i) = \bar{c}_i(z_i) \) for each \( i \in 1..n \). If such a matching \( \theta \) is found, then

1. \textbf{Delete}: We delete \( C_2 \) by setting \( \text{Store} := \text{Store} \setminus C_2 \)
2. \textbf{Create}: We create the body constraints as follows: For the rule body \( B = \{c_{n+1}(z_1), \ldots, c_m(z_m)\} \) we check for a corresponding conjunct \( (b_i \leftrightarrow \theta.\bar{c}_i(z_i)) \) in \( D \) for some \( b_i \). If no such conjunct exists, then we create a new propositional variable \( b \) and set \( D := (b \leftrightarrow \theta.\bar{c}_i(z_i)) \land D \)
3. \textbf{Generate}: Finally we generate clauses that explain the rule application as follows: First we define a function literals() that maps a set of constraints \( \text{Cs} \) to the set of corresponding set of literals as follows:

\[
\text{literals}(\text{Cs}) = \{b_i|c_i(\bar{x}) \in \text{Cs} \} \cup \{-b_i|\neg \bar{c}_i(\bar{x}) \in \text{Cs}\}
\]

where \( b_i \leftrightarrow c_i(\bar{x}) \) is the corresponding conjunct of \( D \). Next we define the set of equality literals \( L_E \), head literals \( L_H \), and body literals \( L_B \) as follows:

\[
L_E = \text{literals}(E) \quad L_H = \text{literals}(C_1 \cup C_2) \quad L_B = \text{literals}(\theta.B)
\]

Finally we generate the following set of clauses:

\[
\text{ChrClauses} := \{-L_E \cup \neg L_H \cup \{l\}| l \in L_B \land \neg \text{true}(l)\}
\]

where \( \neg \{l_1, \ldots, l_n\} \) is shorthand for \( \neg \{l_1, \ldots, l_n\} \) and \( \text{true}(l) \) indicates that literal \( l \) has been set to \text{true}.

The generated clauses \text{ChrClauses} are added to the clause database of the SAT solver. Note that each generated clause is either a unit clause (if \( l \) is unset) or the empty clause (if \( l \) is set to \text{false}). A unit clause will cause \( l \) to be set to \text{true} (via unitPropagate()), which in turn causes chrPropagate() to be reinvoked and the corresponding body constraint to be inserted into the \text{Store}. This may cause further rule application and clause generation. An empty clause will immediately cause failure and backtracking.

\footnote{The answer is “UNKNOWN” since \( s \) satisfies \( B \), and not necessarily \( B \land D \).}
Example 6 (SMCHR Execution). Consider the solver from Example 3 and following goal $G$:

\[
\text{\texttt{leq}(A, B) \land \text{\texttt{leq}}(B, C) \land (\neg \text{\texttt{leq}}(A, C) \lor (A \neq B \land A = C))}
\]

First we normalize $G$ into a propositional formula in CNF and reify CHR constraints as follows

\[
\begin{align*}
[&b_1 \land b_2 \land (\neg b_1 \lor \neg b_2) \land (\neg b_3 \lor b_4)] \land \\
& b_1 \leftrightarrow \text{\texttt{leq}}(A, B) \land b_2 \leftrightarrow \text{\texttt{leq}}(B, C) \land b_3 \leftrightarrow \text{\texttt{leq}}(A, C) \land \\
& b_4 \leftrightarrow (A = B) \land b_5 \leftrightarrow (A = C)
\end{align*}
\]

One possible (simplified) execution tree for $G$ is shown in Figure 1. Here we select literals in order $b_1, \ldots, b_6$. Conflict (failure) states are represented by a cross. The states of interest are:

(A) The start state (no literal set).

(B) After setting $b_1$ and $b_2$, the constraints $\text{\texttt{leq}}(A, B)$ and $\text{\texttt{leq}}(B, C)$ appear in the store. Rule (3) from Example 3 is applied, which generates the clause $\neg b_1 \lor \neg b_2 \lor b_3$.

(C) This branch fails thanks to the clause generated at (B).

(D) After setting $b_1, b_2, \text{ and } b_4$, the constraints $\text{\texttt{leq}}(A, B)$, $\text{\texttt{leq}}(B, C)$, and $A = C$ appear in the store. Rule (2) from Example 3 is applied, which generates the clause $\neg b_1 \lor \neg b_2 \lor b_3 \lor b_5$.

(E) The generated clause from (D) is empty and immediately causes failure.

Since all branches lead to failure, the answer for $G$ is UNSAT, i.e. $G$ is unsatisfiable.

Our method is related to lazy clause generation [Ohri- menko et al., 2009] for finite domain solvers, but generalized to any arbitrary CHR solver. Some SMT solvers, such as Z3 [De Moura and Bjørner, 2008], also support clause generation. For example, see T-Propagate from [Bjørner, 2011].

5 Experiments

Since the original publication in [Duck, 2012], the SMCHR system has been actively developed. We experimentally compare four systems: SMCHR is the latest version (alpha release) of the SMCHR system, Native is a built-in (i.e. non-CHR) bounds propagation solver, SMCHR* is the original prototype implementation from [Duck, 2012], and CHR* is the K.U.Leuven CHR system [Schrijvers and Demoen, 2004] running on SWI Prolog [Wiemeler et al., 2012] version 5.8.2. All timings (in milliseconds) are on Intel i5-2500K CPU clocked at 4GHz and averaged over 10 runs. The results are shown in Figure 1. Here (t.o.) indicates a timeout of 10 minutes, and a dash (–) indicates a benchmark not implemented or not applicable.

<table>
<thead>
<tr>
<th>Bench.</th>
<th>Solver</th>
<th>SMCHR</th>
<th>Native</th>
<th>SMCHR*</th>
<th>CHR*</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle(50)</td>
<td>H</td>
<td>9</td>
<td>–</td>
<td>285</td>
<td>238</td>
</tr>
<tr>
<td>cycle(100)</td>
<td></td>
<td>33</td>
<td>–</td>
<td>8003</td>
<td>3515</td>
</tr>
<tr>
<td>cycle(100)</td>
<td>leq</td>
<td>20</td>
<td>–</td>
<td>501</td>
<td>242</td>
</tr>
<tr>
<td>cycle(100)</td>
<td></td>
<td>105</td>
<td>–</td>
<td>12113</td>
<td>3382</td>
</tr>
<tr>
<td>queens(10)</td>
<td>bounds</td>
<td>861</td>
<td>114</td>
<td>7072</td>
<td>63951</td>
</tr>
<tr>
<td>queens(20)</td>
<td></td>
<td>1144</td>
<td>110</td>
<td>82200</td>
<td>1.0</td>
</tr>
<tr>
<td>subsets(15, 99)</td>
<td></td>
<td>111</td>
<td>46</td>
<td>57</td>
<td>4301</td>
</tr>
<tr>
<td>subsets(20, 99)</td>
<td></td>
<td>105</td>
<td>36</td>
<td>102</td>
<td>79115</td>
</tr>
<tr>
<td>money</td>
<td></td>
<td>380</td>
<td>3</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>zebra</td>
<td></td>
<td>1057</td>
<td>30</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>sudoko</td>
<td></td>
<td>10</td>
<td>3</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>queens(10)</td>
<td>domain</td>
<td>478</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>queens(20)</td>
<td></td>
<td>1272</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>sudoko</td>
<td></td>
<td>210</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>steiner()</td>
<td>set</td>
<td>591</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 1: Experimental results.

6 Conclusions

In this paper we presented Satisfiability Modulo Constraint Handling Rules: the natural merger of SMT with CHR. Our experimental results show that SMCHR is faster than CHR, especially for problems that benefit from no-good learning. Furthermore, the latest version of SMCHR is a significant improvement over the original prototype.

Development of SMCHR is ongoing. Future work includes further optimization, applications, and extensions of the SMCHR system.
References


