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* (CHRs) 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) formulae 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* ($H \Longleftrightarrow B$) that *rewrites* constraints H to B, and *propagation rules* ($H \Longrightarrow B$) 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:*

$$\mathsf{plus}(x,y,z) \wedge \mathsf{lb}(y,l_y) \wedge \mathsf{lb}(z,l_z) \Longrightarrow \mathsf{lb}(x,l_y+l_z)$$
$$\mathsf{plus}(x,y,z) \wedge \mathsf{ub}(y,u_y) \wedge \mathsf{ub}(z,u_z) \Longrightarrow \mathsf{ub}(x,u_y+u_z)$$

Here $\mathsf{plus}(x,y,z)$ *represents* $x = y+z$*,* $\mathsf{lb}(x,l)$ *represents* $x \geq l$*, and* $\mathsf{ub}(x,u)$ *represents* $x \leq u$*. Given an initial goal* $\mathsf{plus}(a,b,c) \wedge \mathsf{lb}(b,3) \wedge \mathsf{ub}(b,10) \wedge \mathsf{lb}(c,4) \wedge \mathsf{ub}(c,6)$*, the rules will* propagate $\mathsf{lb}(a,7) \wedge \mathsf{ub}(a,16)$. *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* $n = 2$*.*

$$(Q_1 = 1 \vee Q_1 = 2) \wedge (Q_2 = 1 \vee Q_2 = 2) \wedge$$
$$\neg(Q_1 = Q_2) \wedge \neg(Q_1 = Q_2 + 1) \wedge \neg(Q_2 = Q_1 + 1)$$

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 Bjørner, 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,

*The paper on which this extended abstract is based was the recipient of the best paper award of the 2012 International Conference on Logic Programming [Duck, 2012].

see [Frühwirth, 1998].

CHR is a rule-based programming language with three types of rules:

$$\begin{aligned} H &\iff B & &(\textit{simplification}) \\ H &\Longrightarrow B & &(\textit{propagation}) \\ H_1 \setminus H_2 &\iff B & &(\textit{simpagation}) \end{aligned}$$

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 $H', E \subseteq S$ where E are *equality constraints*, and a *matching substitution* θ such that $E \rightarrow \theta.H = H'$, then a *simplification rule* $(H \iff B)$ *rewrites* H' to $(\theta.B)$, i.e. $S := (S \backslash H') \cup (\theta.B)$. Likewise, a *propagation rule* $(H \Longrightarrow B)$ *adds* $(\theta.B)$ whilst retaining H', i.e. $S := S \cup (\theta.B)$. Finally a *simpagation rule* $(H_1 \backslash H_2 \Longrightarrow B)$ is a *hybrid* between a simplification and propagation rule, where only the constraints $H_2' \subseteq S$ matching H_2 are rewritten, i.e. $S := (S \backslash H_2') \cup (\theta.B)$. 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:*

$$\mathsf{leq}(X, X) \iff \textit{true} \quad (1)$$

$$\mathsf{leq}(X, Y) \wedge \mathsf{leq}(Y, X) \iff X = Y \quad (2)$$

$$\mathsf{leq}(X, Y) \wedge \mathsf{leq}(Y, Z) \Longrightarrow \mathsf{leq}(X, Z) \quad (3)$$

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

Consider the initial store (a.k.a. the goal*)* $G \equiv \mathsf{leq}(A, B) \wedge \mathsf{leq}(B, C) \wedge \mathsf{leq}(C, A)$*, then one possible execution sequence (a.k.a.* derivation*) is as follows:*

$$\begin{aligned} &\{\underline{\mathsf{leq}(A, B)}, \underline{\mathsf{leq}(B, C)}, \mathsf{leq}(C, A)\} & &\textit{Apply } (3) \\ \rightarrowtail &\{\mathsf{leq}(A, B), \mathsf{leq}(B, C), \underline{\mathsf{leq}(C, A)}, \underline{\mathsf{leq}(A, C)}\} & &\textit{Apply } (2) \\ \rightarrowtail &\{\underline{\mathsf{leq}(A, B)}, \underline{\mathsf{leq}(B, C)}, \underline{A = C}\} & &\textit{Apply } (2) \\ \rightarrowtail &\{A = B, A = C\} \end{aligned}$$

Here the underlined *constraints indicate where the rule is applied. Execution proceeds by first applying the propagation rule (3) which adds the constraint* $\mathsf{leq}(A, C)$*. 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*) $[\![R]\!]$ of a given rule R is defined as follows:

$$\begin{aligned} [\![H \iff B]\!] &= \forall (H \leftrightarrow B) \\ [\![H \Longrightarrow B]\!] &= \forall (H \rightarrow B) \\ [\![H_1 \setminus H_2 \iff B]\!] &= \forall (H_1 \wedge H_2 \leftrightarrow H_1 \wedge B) \end{aligned}$$

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

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 $\forall x, y : x \leq y \wedge y \leq x \leftrightarrow x = y$*, and rule (3) is* transitivity $\forall x, y, z : x \leq y \wedge y \leq z \rightarrow x \leq z$. □

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* leq *is a* total order *relation, then we can extend Example 3 with the following rules defining the negation of* leq*:*

$$\neg\mathsf{leq}(X, Y) \wedge \neg\mathsf{leq}(Y, X) \Longrightarrow \textit{false} \quad (4)$$

$$\neg\mathsf{leq}(X, Y) \wedge \neg\mathsf{leq}(Y, Z) \Longrightarrow \neg\mathsf{leq}(X, Z) \quad (5)$$

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

$$\begin{aligned} &\{\underline{\neg\mathsf{leq}(A, B)}, \underline{\neg\mathsf{leq}(B, C)}\} & &\textit{Apply } (5) \\ \rightarrowtail &\{\neg\mathsf{leq}(A, B), \neg\mathsf{leq}(B, C), \neg\mathsf{leq}(A, C)\} \end{aligned}$$

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

$$\forall x, y : \neg(x \leq y) \wedge \neg(y \leq x) \rightarrow \textit{false}$$

$$\forall x, y, z : \neg(x \leq y) \wedge \neg(y \leq z) \rightarrow \neg(x \leq z) \quad □$$

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 $vars(B) \subseteq vars(H)$. 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:

$$c(\bar{x}) \setminus c(\bar{x}) \iff \textit{true} \quad \text{and} \quad c(\bar{x}) \wedge \neg c(\bar{x}) \Longrightarrow \textit{false}$$

Set-Semantics ensures that each constraint $c(\bar{x})$ 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.

UNSAT means that G is *unsatisfiable* w.r.t. the theory $[\![P]\!]$, i.e. $[\![P]\!] \models \forall\neg G$. The answer UNKNOWN means that SMCHR was unable to prove unsatisfiability. This may be because G is *satisfiable*, or that the solver P is *incomplete* and unable to prove unsatisfiability. This behavior mirrors CHR: if $G \rightarrowtail^* \mathit{false}$, then G is unsatisfiable. Otherwise if $G \rightarrowtail^* S \neq \mathit{false}$, 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* $B \wedge D$ such that:

1. B is a pure *propositional formula* in CNF;
2. D is a conjunction of equivalences of the form $b \leftrightarrow c(\bar{x})$ where b is a *propositional variable* and $c(\bar{x})$ is a constraint; and
3. For all *valuations* (functions mapping variables to values) s there exists a valuation s' such that
$$[\![P]\!] \models s(G) \quad \text{iff} \quad [\![P]\!] \models s'(B \wedge D)$$
and $s(v) = s'(v)$ for all $v \in \mathit{vars}(G)$.

The last condition ensures that $B \wedge D$ is equisatisfiable to G and the solutions of G and $B \wedge D$ 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 $\mathit{Clauses}$ be the set of *clauses* in B. We assume each clause is a set of the form $\{l_1, .., l_n\}$ (representing the disjunction $l_1 \vee .. \vee l_n$), where each l_i is a *literal* b or $\neg b$ for some propositional variable $b \in \mathit{vars}(B)$. The pseudo-code for the DPLL algorithm is as follows:

$$
\begin{array}{l}
\textbf{function}\ \mathsf{dpll}(\mathit{Clauses})\ = \\
\quad \textbf{while}\ \exists b \in \mathit{vars}(\mathit{Clauses}) : \mathsf{unset}(b) \\
\quad\quad l := \mathsf{selectLiteral}(\mathit{Clauses}) \\
\quad\quad \mathit{Clauses} := \mathsf{setLiteral}(\mathit{Clauses}, l) \\
\quad\quad \mathit{Clauses} := \mathsf{unitPropagate}(\mathit{Clauses}) \\
\quad\quad \textbf{if}\ \emptyset \in \mathit{Clauses} \\
\quad\quad\quad \mathit{Clauses} := \mathsf{backtrack}(\mathit{Clauses}) \\
\quad\quad\quad \textbf{if}\ \mathit{Level} = 0\ \ \textbf{return}\ \mathsf{UNSAT} \\
\quad \textbf{return}\ \mathsf{UNKNOWN}
\end{array}
$$

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 $\mathit{vars}(B)$ that satisfies B, and the answer UNKNOWN is returned.[1] Our pseudo-code is a simplification. An actual SMCHR implementation will typically use a modern SAT solver design [Een and Srensson, 2003] with conflict-driven search, no-good learning, back-jumping, etc.

[1]The answer is "UNKNOWN" since s satisfies B, and not necessarily $B \wedge D$.

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

$$
\begin{array}{l}
\textbf{function}\ \mathsf{chrPropagate}(l, \mathit{Clauses})\ = \\
\quad \textbf{if}\ D \equiv (b \leftrightarrow c(\bar{x}) \wedge ..) \\
\quad\quad \mathit{Store} := \mathit{Store} \cup \{(l \equiv b?\ c(\bar{x}) :\ \neg c(\bar{x})\} \\
\quad\quad \textbf{let}\ \mathit{ChrClauses} = \mathsf{chrMatch}(\mathit{Rules}, \mathit{Store}) \\
\quad\quad \mathit{Clauses} := \mathit{Clauses} \cup \mathit{ChrClauses} \\
\quad \textbf{return}\ \mathit{Clauses}
\end{array}
$$

If l (via b) corresponds to a constraint $c(\bar{x})$, then the Store is updated to include $c(\bar{x})$ if $l \equiv b$ or $\neg c(\bar{x})$ if $l \equiv \neg b$. Next the chrMatch routine attempts to match (and apply) a CHR rule. We shall use the notation $\hat{c}(\bar{x})$ to represent a *constraint literal* $c(\bar{x})$ or $\neg c(\bar{x})$. The chrMatch routine searches for a (renamed apart) rule $(H_1 \setminus H_2 \Longleftrightarrow B) \in$ Rules and sets of constraints $C_1, C_2, E \subseteq \mathit{Store}$ such that E is (a minimal set of) equality constraints, and for $C_1 \uplus C_2 = \{\hat{c}_1(\bar{x}_1), .., \hat{c}_n(\bar{x}_n)\}$ and $H_1 \uplus H_2 = \{\hat{c}_1(\bar{y}_1), .., \hat{c}_n(\bar{y}_n)\}$ there exists a *matching substitution* θ such that $E \rightarrow \theta.\hat{c}_i(\bar{y}_i) = \hat{c}_i(\bar{x}_i)$ for each $i \in 1..n$. If such a matching θ is found, then

1. **Delete**: We *delete* C_2 by setting $\mathit{Store} := \mathit{Store} - C_2$
2. **Create**: We *create* the body constraints as follows: For the rule body $B = \{\hat{c}_{n+1}(\bar{z}_1), .., \hat{c}_m(\bar{z}_m)\}$ we check for a corresponding conjunct $(b_i \leftrightarrow \theta.c_i(\bar{z}_i))$ in D for some b_i. If no such conjunct exists, then we create a new propositional variable b and set $D := (b \leftrightarrow \theta.c_i(\bar{z}_i)) \wedge D$
3. **Generate**: Finally we *generate* clauses that *explain* the rule application as follows: First we define a function literals() that maps a set of constraints Cs to the set of corresponding set of literals as follows:
$$\mathsf{literals}(Cs) = \{b_i | c_i(\bar{x}) \in Cs\} \cup \{\neg b_i | \neg c_i(\bar{x}) \in Cs\}$$
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:
$$L_E = \mathsf{literals}(E) \quad L_H = \mathsf{literals}(C_1 \uplus C_2)$$
$$L_B = \mathsf{literals}(\theta.B)$$
Finally we *generate* the following set of clauses:
$$\mathit{ChrClauses} = \{\neg L_E \cup \neg L_H \cup \{l\} | l \in L_B \wedge \neg\mathsf{true}(l)\}$$
where $\neg\{l_1, .., l_n\}$ is shorthand for $\{\neg l_1, .., \neg l_n\}$ and $\mathsf{true}(l)$ indicates that literal l has been set to true.

The generated clauses $\mathit{ChrClauses}$ 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 *false*). A unit clause will cause l to be set to true (via unitPropagate()), which in turn causes chrPropagate() to be reinvoked and the corresponding body constraint to be inserted into the Store. This may cause further rule application and clause generation. An empty clause will immediately cause failure and backtracking.

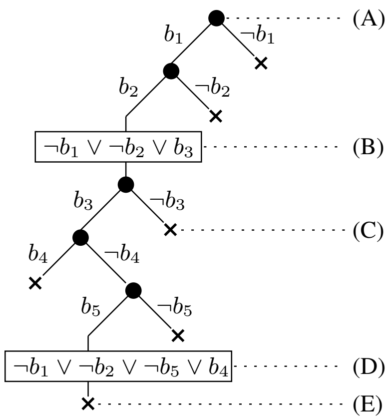

Figure 1: SMCHR execution tree.

Example 6 (SMCHR Execution)**.** *Consider the solver from Example 3 and following goal G:*

$$\mathsf{leq}(A,B) \wedge \mathsf{leq}(B,C) \wedge (\neg\mathsf{leq}(A,C) \vee (A \neq B \wedge A = C))$$

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

$$\begin{gathered}\big[b_1 \wedge b_2 \wedge (\neg b_3 \vee \neg b_4) \wedge (\neg b_3 \vee b_5)\big] \wedge \\ b_1 \leftrightarrow \mathsf{leq}(A,B) \wedge b_2 \leftrightarrow \mathsf{leq}(B,C) \wedge b_3 \leftrightarrow \mathsf{leq}(A,C) \wedge \\ b_4 \leftrightarrow (A = B) \wedge b_5 \leftrightarrow (A = C)\end{gathered}$$

One possible (simplified) execution tree for G is shown in Figure 1. Here we select literals in order $b_1, .., b_5$. 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 $\mathsf{leq}(A,B)$ *and* $\mathsf{leq}(B,C)$ *appear in the* store. *Rule (3) from Example 3 is applied, which generates the clause $(\neg b_1 \vee \neg b_2 \vee b_3)$.*
(C) This branch fails thanks to the clause generated at (B).
(D) After setting b_1, b_2, and b_5, the constraints $\mathsf{leq}(A,B)$*,* $\mathsf{leq}(B,C)$*, and $A = C$ appear in the* store. *Rule (2) from Example 3 is applied, which generates the clause $(\neg b_1 \vee \neg b_2 \vee \neg b_5 \vee b_4)$.*
(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* [Ohrimenko *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 [Wielemaker *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.

The `cycle`(n), `queens`(n), and `subsets`(n, v) benchmarks are from [Duck, 2012]. The remaining benchmarks are: `money` is the send+more=money crypto-arithmetic puzzle, `sudoku` is a Sudoku instance, `zebra` is the zebra puzzle, and `steiner`(n) computes Steiner triples. The CHR solvers are: leq, lt are the solvers from Examples 3 and 5 respectively, bounds is a bounds propagation solver (see Example 1), domain is a domain propagation solver, and set is a set solver. These solvers are distributed with the SMCHR system.

The no-good learning and search space pruning of SMCHR (both versions) give SMCHR a clear advantage over Prolog CHR for benchmarks that use search. The latest version of SMCHR is a clear improvement over the original implementation. This is mainly because of several new optimizations, such as better search heuristics and hash-table based indexing for the constraint *store*. Finally, the native implementation of a bounds propagation solver is faster than the SMCHR implementation. This is in line with expectations: a solver implemented in CHR is generally slower than a native implementation, as CHR is generally a trade-off between implementation effort vs. solver speed. Nevertheless, the results suggest that there is room for further optimization.

		SMCHR	Native	SMCHR*	CHR*
Bench.	Solver	time (ms)			
cycle(50)	lt	9	–	285	238
cycle(100)	$''$	33	–	8003	3515
cycle(50)	leq	20	–	501	242
cycle(100)	$''$	105	–	12113	3382
queens(16)	bounds	861	114	7072	65951
queens(20)	$''$	1141	110	82200	t.o.
subsets$(15, 99)$	$''$	111	46	57	4301
subsets$(20, 99)$	$''$	105	36	102	79115
money	$''$	380	3	–	–
sudoku	$''$	1057	30	–	–
zebra	$''$	10	3	–	–
queens(16)	domain	478	–	–	–
queens(20)	$''$	1272	–	–	–
sudoku	$''$	210	–	–	–
steiner(7)	set	591	–	–	–

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

[Bjørner, 2011] N. Bjørner. Engineering theories with Z3. In *Proceedings of the First international conference on Certified Programs and Proofs*. Springer-Verlag, 2011.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[De Moura and Bjørner, 2008] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[Duck *et al.*, 2004] G. Duck, P. Stuckey, M. Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In *International Conference on Logic Programming*, pages 90–104. Springer, 2004.

[Duck, 2012] G. Duck. SMCHR: Satisfiability modulo constraint handling rules. *Theory and Practice of Logic Programming*, 12(4-5):601–618, 2012. Proceedings of the 28^{th} international conference on Logic Programming.

[Een and Srensson, 2003] N. Een and N. Srensson. An extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing*. Springer Verlag, 2003.

[Frühwirth, 1998] T. Frühwirth. Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming, Journal of Logic Programming*, 37, October 1998.

[Moura and Bjørner, 2011] L. Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communcations of the ACM*, 54(9):69–77, September 2011.

[Ohrimenko *et al.*, 2009] O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009.

[Schrijvers and Demoen, 2004] T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.

[Sneyers *et al.*, 2010] J. Sneyers, P. Weert, T. Schrijvers, and L. Koninck. As time goes by: Constraint handling rules – a survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming*, 10:1–47, 2010.

[Wielemaker *et al.*, 2012] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.