Compiling Constraint Handling Rules for Efficient Tabled Evaluation*

Beata Sarna-Starosta[1] and C. R. Ramakrishnan[2]

[1] Dept. of Comp. Sci. & Engg., Michigan State University, East Lansing, MI 48824
E-mail: bss@cse.msu.edu
[2] Dept. of Computer Science, University at Stony Brook, Stony Brook, NY 11794
E-mail: cram@cs.sunysb.edu

Abstract. Tabled resolution, which alleviates some of Prolog's termination problems, makes it possible to create practical applications from high-level declarative specifications. Constraint Handling Rules (CHR) is an elegant framework for implementing constraint solvers from high-level specifications, and is available in many Prolog systems. However, applications combining the power of these two declarative paradigms have been impractical since traditional CHR implementations interact poorly with tabling. In this paper we present a new (set-based) semantics for CHR which enables efficient integration with tabling. The new semantics coincides with the traditional (multi-set-based) semantics for a large class of CHR programs. We describe CHRd, an implementation based on the new semantics. CHRd uses a distributed constraint store that can be directly represented in tables. Although motivated by tabling, CHRd works well also on non-tabled platforms. We present experimental results which show that, relative to traditional implementations, CHRd performs significantly better on tabled programs, and yet shows comparable results on non-tabled benchmarks.

1 Introduction

Constraint Logic Programming (CLP) is an elegant framework for encoding a wide variety of problems ranging from infinite-state system verification [6, 7] to specification and analysis of security policies [3, 15]. However, traditional CLP systems are unsuitable for directly evaluating these formulations since they use Prolog-style resolution strategy, and, consequently, inherit Prolog's weak termination (infinite looping) and efficiency (repeated subcomputations) problems. Tabled resolution [4, 28] overcomes these problems by memoizing subgoals and computed answers during resolution, and reusing them. Prolog systems enhanced with tabling (e.g. XSB [19]) have supported the construction of efficient tools for program analysis and the verification of finite state systems [5, 18] based on high-level logical specifications. Combining constraint processing with tabled resolution will enable evaluating complex applications, such as the analysis of infinite state systems, directly from high-level specifications.

Constraint Handling Rules (CHR) is a rule-based committed-choice language that is particularly well-suited for specifying constraint solvers at a high level [10]. CHR

* This research was supported in part by NSF grants CCR-0205376, CNS-0627447 and EIA-0000433, and ONR grant N00014-01-1-0744.

has been implemented in a variety of Prolog systems including SICStus [27], and hProlog [8]. The lack of tabled CLP systems was addressed by the recent port of hProlog's CHR to XSB [24] (called XSB-CHR in the remainder of this paper). However, as explained below, the data structures and algorithms used in traditional CHR systems are unsuitable for use with tabled resolution, leading to severe performance problems in XSB-CHR. This paper describes CHRd, an alternative implementation of CHR, that in addition to working with traditional Prolog systems, seamlessly integrates CHR with tabling. The efficiency of CHRd permits high-level implementations of applications combining constraint solving and tabling.

Background. Operationally, CHR programs can be viewed as rewriting rules. The constraint store is a multi-set of constraints, and the rules specify how the store should evolve. For instance, consider the CHR program for the partial order constraint:

Example 1

```
reflexivity  @ leq(X,X)             <=> true.
idempotence  @ leq(X,Y) \ leq(X,Y) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X)  <=> X=Y.
transitivity @ leq(X,Y), leq(Y,Z)  ==> leq(X,Z).
```

Above, `reflexivity` and `antisymmetry` are *simplification* rules. The latter states that every pair of constraints in the store that match `leq(X,Y)` and `leq(Y,X)` should be *replaced* by the equality constraint `X=Y` (a built-in constraint solved by unifying `X` and `Y`). `transitivity` is a *propagation* rule. It states that for every pair of constraints that match the left hand side, the corresponding right hand side constraint should be *added* to the store. Since the constraint store is a multi-set, it may contain more than one instance of the same constraint. The *simpagation* rule `idempotence` (which combines simplification and propagation) ensures that the store is a set. It states that in the presence of one instance of `leq(X,Y)` (to the left of '`\`') another instance of `leq(X,Y)` should be replaced by `true` (i.e. removed from the store).

A rule becomes applicable when the store contains the constraints that match its left hand side. CHR evaluation proceeds by repeatedly selecting and firing an applicable rule (i.e. forward chaining) until no rule is applicable (i.e. a fixed point is reached).

Note that a propagation rule remains applicable even after it has been fired. Since the constraint store is a multi-set, re-firing a propagation rule will change the store, adding new copies of constraints. To avoid trivial nontermination due to firing the same propagation rule over and over again, the CHR operational semantics (and, subsequently, its implementations) maintain *propagation history*, a record of all instances of propagation rules that have been fired so far. A rule is applicable only if its instance is not in the propagation history.

Traditional CHR and Tabling. The idea of tabling is to record subgoals (calls) and their provable instances (answers) so that the results of a computation done in one context can be re-used in another. When tabling is integrated with constraint processing, we need to associate a constraint store with each call and answer to properly record the context of a computation. This leads to several efficiency problems. First, the CHR constraint store as well as its propagation history needs to be copied in and out of tables; traditional CHR representation of constraints (with their cyclic terms) are not well-suited for storage in tables. Second, as shown in [24], storing the propagation history imposes a heavy space burden, but not storing it leads to very high time overheads for re-propagating the constraints when they are retrieved from tables. Thus a port of a

traditional CHR implementation to a tabled environment (represented by XSB-CHR) imposes significant performance penalties on tabled applications.

Our Solution. We combine CHR evaluation and tabling by taking a fundamentally different approach to CHR. We give CHR a set-based semantics that addresses the trivial nontermination problem *without the use of propagation history*. The new semantics is formulated so as to coincide with CHR's well-accepted semantics [9] for a large class of programs (see Section 3). In our implementation, called CHRd, we consider a syntactically restricted class called direct-indexed CHR, where all constraint terms in every rule head are connected by common variables. This class covers a large number of CHR-based constraint solvers. The restriction permits the constraint store to be represented in a distributed fashion, as a network of constraints on the individual variables (see Section 4). The distributed store and the absence of propagation history enables direct representation of constraint stores in tables, significantly reducing the time taken to switch between constraint stores in tabled evaluation. Our implementation has been integrated into XSB v3.0.1, and the latest version can be obtained from XSB's CVS repository at `http://xsb.sourceforge.net`.

CHRd enables us to efficiently evaluate applications that combine tabled evaluation and constraint processing, and to scale up to problem sizes of practical importance. A case in point is an application for the analysis of concurrent object-oriented systems based on a high-level formulation in terms of CHR rules and tabled logic programs [21]. The relatively good performance of CHRd is crucial to the success of this application. Moreover, CHRd itself is independent of tabling; its performance is comparable to that of existing CHR implementations on non-tabled platforms[1]. A detailed description of the experimental results appears in Section 5.

It should also be noted that *ground CHR*, a class that is of significant interest to the CHR community, is not direct-indexed. Nevertheless, ground CHR programs can be readily converted into programs that can be evaluated by CHRd (Section 4). Moreover, many of the recently developed CHR optimizations (e.g. selection of indexing structures) are valid for CHRd. We discuss the relationship between this paper and previous work on CHR and its implementations in Section 6.

2 Preliminaries

We use standard notions of variables, terms and substitutions [16]. We use t to refer to terms in general, c for *constraint* terms which have a constraint symbol as root, and b for built-in constraint terms which have a built-in constraint symbol as root. We use $\mathit{vars}(t)$ to refer to the set of variables in the term t. We write $\uplus$ to represent disjoint union, and $+\!\!+$ to denote concatenation of ordered sequences. Sets and multi-sets are occasionally considered as sequences with non-deterministically chosen order of elements. Substitutions are denoted by θ, and a term t under θ is written as $t\theta$. We use upper-case letters such as G, S, etc. to denote collections (sets, multi-sets or sequences) and lower-case letters for elements of these collections.

CHR Syntax. A CHR program is a finite set of rules that specify how *user-defined* constraints are solved based on the host language's *built-in* constraints (e.g. Prolog pred-

[1] The CHRd system for other platforms including hProlog and SWI-Prolog is available at `http://www.cse.msu.edu/~bss/chrd`.

icates). CHR rules are of the form:

$$\textit{label} \ @ \ \textit{Head} \left\{ \begin{array}{c} <=> \\ ==> \end{array} \right\} \textit{Guard} \mid \textit{Body}$$

Simpagation rules are the most general. They are of the form $H_1 \setminus H_2$ <=> $G \mid B$ where H_1 and H_2 are sequences of user-defined constraint terms (the *heads* of the rule), G (the *guard*) is a sequence of built-in constraints and B (the *body*) is a sequence of built-in and user-defined constraint terms. A rule specifies that when constraints in the store match H_1 and H_2 and the guard G holds, the constraints that match H_2 can be *replaced* by the corresponding constraints in B. The literal `true` represents an empty sequence of constraint terms. The guard part, $G \mid$, may be omitted when G is empty.

A simplification rule, which has the form H_2 <=> $G \mid B$ can be represented by a simpagation rule `true` $\setminus H_2$ <=> $G \mid B$. Similarly, a propagation rule, which has the form H_1 ==> $G \mid B$, can be represented by a simpagation rule $H_1 \setminus$ `true` <=> $G \mid B$.

CHR Semantics. CHR has a well-defined declarative as well as operational semantics [1, 10]. The declarative interpretation of a CHR program P is given by the set of universally quantified formulas corresponding to the CHR rules, and an underlying consistent constraint theory *CT*. The constraint theory defines the meaning of host language constraints, the equality constraint '=', and the boolean atoms *true* and *false*.

The original operational semantics [1] is given in terms of a non-deterministic transition system. The evaluation of a program P is a path through the transition system. The transitions are made when a constraint is added from the goal to the store, or by firing any applicable program rule. The refined semantics ω_r [9] defines a more deterministic transition system, specifying, among others, the order in which rules are tried. Most CHR implementations are based on ω_r.

3 The Set-Based Operational Semantics

Our set-based operational semantics, called ω_{set}, is given in terms of a transition relation. The formulation of ω_{set} closely follows that of the refined operational semantics ω_r [9]. A state in the system is represented by a triple $\langle E, C_U, C_B \rangle_{\mathcal{V},P}$ where E, called the *execution stack*, is an ordered sequence of constraint activation events; C_U, the user-defined constraint store, is a set of user-defined constraints, and C_B, the built-in constraint store, is a conjunction of built-in constraints; $\mathcal{V}$ is a sequence of variables; and P is the given CHR program. We omit either one or both the subscripts $\mathcal{V}, P$ whenever clear from the context. In contrast, states in ω_r are quadruples $\langle E, C_U, C_B, \mathcal{T} \rangle_{\mathcal{V},P}$ where $\mathcal{T}$ is the propagation history and C_U, the user-defined store, is a multi-set.

As in the refined operational semantics ω_r, different occurrences of constraint terms with the same symbol in the heads of rules are marked with an *occurrence* number corresponding to the order in which they appear in the CHR program (starting from 1). This numbering indicates the order in which the rules are tried, thus reducing non-determinism of program evaluation. Three kinds of activation events can appear in the execution stack:

- Inactive constraint: c is a user-defined or built-in constraint term;

– Active constraint: $c\!:\!j$ where c is a user-defined constraint term and j is a number, meaning that this term can match only with the j-th occurrence of the constraint symbol in the program; and
– Conditional activation: $(H_1 \backslash H_2), G \triangleright B$ where H_1 and H_2 are sets of user-defined constraint terms, G is a set of built-in constraint terms, and B is a sequence of user-defined and built-in constraint terms.

A CHR program is evaluated by forward chaining, and the evaluation stack is used to control this computation. The event at the top of the execution stack is the one *currently scheduled* for evaluation. The first two events above are also in ω_r; the conditional activation event is unique to ω_{set}[2]. An inactive constraint (the first event) corresponds to constraints that we have not yet begun processing; an active constraint corresponds to one that is being processed. The conditional activation event marks constraints that we will begin processing only when the conditions hold.

The initial state of the system is $\langle E, \emptyset, \mathit{true}\rangle_{\mathcal{V},P}$ where E is the sequence of constraints posed to the system, $\mathcal{V}$ is the set of variables in E, P is the CHR program. A successful terminating state of the system is of the form $\langle \top, C_U, C_B\rangle$, where $\top$ is an empty execution stack and $C_B \neq \mathit{false}$. A failed state is one where $C_B = \mathit{false}$. The *logical* reading of a state $\langle E, C_U, C_B\rangle_{\mathcal{V},P}$ is $\exists \bar{x}\ E \wedge C_U \wedge C_B$ where E and C_U denote conjunctions of their respective contents and $\bar{x}$ is the set of variables in the state that are not in $\mathcal{V}$.

Note that a rule that was not applicable when a constraint was initially activated may become applicable when variables in that constraint are bound. We determine which constraints need to be reprocessed using the *wakeup* function defined below. We say that a built-in constraint b *affects* a user-defined constraint c in store C_U (denoted by $c \in \mathit{affects}(b, C_U)$) if the evaluation of b adds bindings to any variable in c. A variable x is *fixed* in the built-in store C_B (denoted by $x \in \mathit{fixed}(C_B)$) if there is only one value for x that makes C_B true. A constraint c in C_U is *fixed* in C_B (denoted by $c \in \mathit{fixed}(C_B, C_U)$) if $\mathit{vars}(c) \subseteq \mathit{fixed}(C_B)$. We need to reprocess the set of all constraints in C_U that are affected by b but are not fixed by C_B. Since these constraints have been activated before, we define the wakeup function to directly generate active constraints. Formally, $\mathit{wakeup}(b, C_U, C_B) = \{c\!:\!1 \mid c \in \mathit{affects}(b, C_U) \wedge c \notin \mathit{fixed}(C_U, C_B)\}$.

3.1 Derivation Rules for ω_{set}

Derivations of ω_{set} are given by the relation $\mapsto_{set}$ which defines transitions of the form $\sigma_{src} \mapsto_{set} \sigma_{dst}$ according to the following rules. An example ω_{set} derivation for the `leq` program from Example 1 is shown in Fig. 1. We illustrate application of rules of $\mapsto_{set}$ with appropriate transitions in this derivation.

Activate : Let $\sigma_{src} = \langle [c|E], C_U, C_B\rangle$ and $c \notin C_U$. That is, c is an inactive user-defined constraint that is not already in the store. Then c is added to the CHR store and annotated to match its first occurrence in P. That is, $\sigma_{dst} = \langle [c\!:\!1|E], \{c\} \cup C_U, C_B\rangle$.

For example, the **Activate** transition in Fig. 1, lines (1–2), sets the currently scheduled constraint `leq(A,B)` to match the first occurrence of `leq` in the program, and adds it to the constraint store.

[2] It should be noted that while formal definition of ω_r does not have conditional activation, most CHR *implementations* use this notion implicitly [12, 22].

$$\langle[\texttt{leq(A,B)}, \texttt{leq(B,C)}, \texttt{leq(A,C)}, \texttt{leq(C,A)}], \emptyset, \mathit{true}\rangle \quad (1)$$

$$\mathbf{Activate} \mapsto_{\mathit{set}} \langle[\texttt{leq(A,B)}:1, \texttt{leq(B,C)}, \texttt{leq(A,C)}, \texttt{leq(C,A)}], \{\texttt{leq(A,B)}\}, \mathit{true}\rangle \quad (2)$$

$$7{*}\mathbf{Default} \mapsto_{\mathit{set}} \langle[\texttt{leq(A,B)}:8, \texttt{leq(B,C)}, \texttt{leq(A,C)}, \texttt{leq(C,A)}], \{\texttt{leq(A,B)}\}, \mathit{true}\rangle \quad (3)$$

$$\mathbf{Drop}^{>} \mapsto_{\mathit{set}} \langle[\texttt{leq(B,C)}, \texttt{leq(A,C)}, \texttt{leq(C,A)}], \{\texttt{leq(A,B)}\}, \mathit{true}\rangle \quad (4)$$

$$\begin{matrix}\mathbf{Activate}\\ 6{*}\mathbf{Default}\end{matrix} \mapsto_{\mathit{set}} \langle[\texttt{leq(B,C)}:7, \texttt{leq(A,C)}, \texttt{leq(C,A)}], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}\}, \mathit{true}\rangle \quad (5)$$

$$\mathbf{PropMatch} \mapsto_{\mathit{set}} \langle[(\texttt{leq(A,B)}, \texttt{leq(B,C)} \backslash \emptyset), \mathit{true} \triangleright \texttt{leq(A,C)}, \texttt{leq(B,C)}:8, \texttt{leq(A,C)}, \texttt{leq(C,A)}], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}\}, \mathit{true}\rangle \quad (6)$$

$$\mathbf{PropFire} \mapsto_{\mathit{set}} \langle[\texttt{leq(A,C)}, \texttt{leq(B,C)}:8, \texttt{leq(A,C)}, \texttt{leq(C,A)}], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}\}, \mathit{true}\rangle \quad (7)$$

$$\begin{matrix}\mathbf{Activate}\\ 7{*}\mathbf{Default}\\ \mathbf{Drop}^{>}\end{matrix} \mapsto_{\mathit{set}} \langle[\texttt{leq(B,C)}:8, \texttt{leq(A,C)}, \texttt{leq(C,A)}], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}, \texttt{leq(A,C)}\}, \mathit{true}\rangle \quad (8)$$

$$\mathbf{Drop}^{>} \mapsto_{\mathit{set}} \langle[\texttt{leq(A,C)}, \texttt{leq(C,A)}], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}, \texttt{leq(A,C)}\}, \mathit{true}\rangle \quad (9)$$

$$\mathbf{Drop}^{<} \mapsto_{\mathit{set}} \langle[\texttt{leq(C,A)}], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}, \texttt{leq(A,C)}\}, \mathit{true}\rangle \quad (10)$$

$$\begin{matrix}\mathbf{Activate}\\ 3{*}\mathbf{Default}\end{matrix} \mapsto_{\mathit{set}} \langle[\texttt{leq(C,A)}:4], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}, \texttt{leq(A,C)}\}, \mathit{true}\rangle \quad (11)$$

$$\mathbf{Simplify} \mapsto_{\mathit{set}} \langle[\texttt{C} = \texttt{A}], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}\}, \mathit{true}\rangle \quad (12)$$

$$\mathbf{Solve} \mapsto_{\mathit{set}} \langle[\texttt{leq(A,B)}:1, \texttt{leq(B,C)}:1], \{\texttt{leq(A,B)}, \texttt{leq(B,C)}\}, \texttt{C} = \texttt{A}\rangle \quad (13)$$

$$\begin{matrix}3{*}\mathbf{Default}\\ \mathbf{Simplify}\end{matrix} \mapsto_{\mathit{set}} \langle[\texttt{C} = \texttt{B}, \texttt{leq(B,C)}:1], \{\texttt{leq(A,B)}\}, \texttt{C} = \texttt{A}\rangle \quad (14)$$

$$\mathbf{Solve} \mapsto_{\mathit{set}} \langle[\texttt{leq(A,B)}:1, \texttt{leq(B,C)}:1], \{\texttt{leq(A,B)}\}, \texttt{C} = \texttt{A} \wedge \texttt{C} = \texttt{B}\rangle \quad (15)$$

$$\mathbf{Simplify} \mapsto_{\mathit{set}} \langle[\texttt{leq(B,C)}:1], \emptyset, \texttt{C} = \texttt{A} \wedge \texttt{C} = \texttt{B}\rangle \quad (16)$$

$$\begin{matrix}7{*}\mathbf{Default}\\ \mathbf{Drop}^{>}\end{matrix} \mapsto_{\mathit{set}} \langle[\,], \emptyset, \texttt{C} = \texttt{A} \wedge \texttt{C} = \texttt{B}\rangle \quad (17)$$

Fig. 1. Derivation for the `leq` program under ω_{set}

Default : If no other transition can be fired in a state $\sigma_{\mathit{src}} = \langle[c\!:\!j|E], C_U, C_B\rangle$, then the currently scheduled constraint $c\!:\!j$ is assigned the next occurrence number. That is, $\sigma_{\mathit{dst}} = \langle[c\!:\!j{+}1|E], C_U, C_B\rangle$.

For example, each of the seven **Default** transitions in Fig. 1, lines (2–3), increments the occurrence index `j` of the currently scheduled constraint $\texttt{leq(A,B)} : \texttt{j}$ until the occurrence number is 8. Since there are only seven occurrences of `leq` in the program, this enables the **Drop**$^{>}$rule.

Drop$^{>}$: Let $\sigma_{\mathit{src}} = \langle[c\!:\!j|E], C_U, C_B\rangle$ where c does not have a j-th occurrence in P (i.e., all occurrences of c have been tried with the **Default** rule; see below). Then $c\!:\!j$ is popped from the execution stack. That is, $\sigma_{\mathit{dst}} = \langle E, C_U, C_B\rangle$.

For example, the **Drop**$^{>}$transition in Fig. 1, lines (3–4), pops $\texttt{leq(A,B)}:8$ from the execution stack as there are only seven occurrences of `leq` in the program.

PropMatch : Let $\sigma_{\mathit{src}} = \langle[c\!:\!j|E], C_U, C_B\rangle$, the program P contain rule $R = c' : j, H_1' \backslash H_2' <=> G \mid B$. Also let θ be a substitution s.t. $c'\theta = c$, $H_1'\theta$, $H_2'\theta$ and $\{c'\theta\}$ are all mutually disjoint subsets of C_U, and $\mathit{CT} \models C_B \rightarrow \exists\bar{x}(G\theta)$ where $\bar{x}$ are variables that occur in G but not in C_B. That is, there is a substitution under which the

constraint store matches the heads of rule R and satisfies its guard. Then the currently scheduled constraint is assigned the next occurrence number. Moreover, *all* matching substitutions are computed iteratively, and the body constraints of R under these substitutions are pushed onto the stack. Note, however, that before a body constraint thus pushed on the stack is taken up for evaluation, some of the constraints used in the match may be removed from the store. Hence we create *conditional activations* for the body constraints. Formally, let $\{\theta_1, \ldots, \theta_n\}$ be the set of all most general substitutions such that $c'\theta_i = c$, $H'_1\theta_i$, $H'_2\theta_i$ and $\{c'\theta_i\}$ are all mutually disjoint subsets of C_U, and $\boldsymbol{CT} \models C_B \rightarrow \exists\bar{x}(G\theta_i)$. Let $\Gamma_i = (H'_1\theta_i \backslash H'_2\theta_i), G\theta_i \triangleright B\theta_i$. Then, $\sigma_{dst} = \langle[\Gamma_1, \ldots, \Gamma_n] +\!\!+ [c\!:\!j{+}1|E], C_U, C_B\rangle$.

For example, the **PropMatch** transition in Fig. 1, lines (5–6), matches the stored constraint $\texttt{leq}(\texttt{A},\texttt{B})$ and the currently scheduled constraint $\texttt{leq}(\texttt{B},\texttt{C}):7$ with the head of the `transitivity` rule in the program. The occurrence index of the currently scheduled constraint is incremented by 1, and the corresponding body constraint $\texttt{leq}(\texttt{A},\texttt{C})$, annotated with the matched head constraints, is pushed onto the execution stack.

PropFire : Let $\sigma_{src} = \langle[(H_1 \backslash H_2), G \triangleright B|E], H_1 \uplus H_2 \uplus C_U, C_B\rangle$, such that $\boldsymbol{CT} \models C_B \rightarrow \exists\bar{x}(G)$ where $\bar{x}$ are variables that occur in G but not in C_B. That is, a conditional activation event is on top of the stack such that the constraints in $H_1 \uplus H_2$ exist in the user-defined store, and the guard G is satisfied by the built-in store. Then the constraints in H_2 are removed from the user-defined store, and all constraints in B are pushed onto the evaluation stack. Formally, $\sigma_{dst} = \langle B +\!\!+ E, H_1 \uplus C_U, C_B\rangle$.

For example, the **PropFire** transition in Fig. 1, lines (6–7), verifies that the constraints $\texttt{leq}(\texttt{A},\texttt{B})$ and $\texttt{leq}(\texttt{B},\texttt{C})$, which matched the head of the `transitivity` rule and caused pushing $\texttt{leq}(\texttt{A},\texttt{C})$ onto the execution stack, are present in the constraint store C_U, and schedules $\texttt{leq}(\texttt{A},\texttt{C})$ for evaluation.

PropDrop : Let $\sigma_{src} = \langle[(H_1 \backslash H_2), G \triangleright B|E], C_U, C_B\rangle$ such that either $(H_1 \uplus H_2) \not\subseteq C_U$ or $\boldsymbol{CT} \not\models C_B \rightarrow \exists\bar{x}(G)$ where $\bar{x}$ are variables that occur in G but not in C_B. That is, a conditional activation event is on top of the stack, and its condition is not satisfied. Then the currently scheduled event is popped from the stack: $\sigma_{dst} = \langle E, C_U, C_B\rangle$.

Drop$^<$: Let $\sigma_{src} = \langle[c|E], C_U, C_B\rangle$ and $c \in C_U$. That is, c is an inactive constraint that is already in the store. Then c is popped from the execution stack: $\sigma_{dst} = \langle E, C_U, C_B\rangle$.

For example, the **Drop**$^<$transition in lines (9–10) of Fig. 1 pops $\texttt{leq}(\texttt{A},\texttt{C})$ from the execution stack since it is already in the constraint store.

Simplify : Let $\sigma_{src} = \langle[c\!:\!j|E], \{c\} \uplus H_1 \uplus H_2 \uplus C_U, C_B\rangle$ and the program P contain a matching rule R. I.e., $R = H'_1 \backslash c' : j, H'_2 \mathrel{<\!\!=\!\!>} G \mid B$ and there is a substitution θ s.t. $H'_1\theta = H_1$, $H'_2\theta = H_2$, $c'\theta = c$, and $\boldsymbol{CT} \models C_B \rightarrow \exists\bar{x}(G\theta)$ where $\bar{x}$ are variables that occur in G but not in C_B. Then $c\!:\!j$ is popped from the execution stack, all constraints matching H'_2 are removed from the store, and R's body constraints under the substitution θ are pushed onto the stack. Formally, $\sigma_{dst} = \langle B\theta +\!\!+ E, H_1 \uplus C_U, C_B\rangle$

For example, the **Simplify** transition in Fig. 1 lines (11–12), matches the active constraint with $4th$ occurrence of `leq` (i.e. the `antisymmetry` rule), removes $\texttt{leq}(\texttt{A},\texttt{C})$, and adds the rule body $\texttt{C} = \texttt{A}$ to the stack.

Solve : Let $\sigma_{src} = \langle[b|E], C_U, C_B\rangle$ and b be a built-in constraint. Then b is added to the built-in store and all constraints affected by b but not fixed by C_B are pushed onto the execution stack. Formally, $\sigma_{dst} = \langle \mathit{wakeup}(b, C_U, C_B) +\!\!+ E, C_U, b \wedge C_B\rangle$.

For example, the **Solve** transition in Fig. 1, lines (14–15), processes the scheduled constraint $\texttt{C} = \texttt{B}$, by first adding $\texttt{C} = \texttt{B}$ to the built-in store. The affected constraint $\texttt{leq}(\texttt{A}, \texttt{B})$ is re-activated, and made the new scheduled constraint.

The transitions **PropMatch**, **PropFire** and **PropDrop** directly correspond to the way constraint propagation is implemented. The universal search eliminates the problem of trivial nontermination due to repeated firing of a propagation rule for the same active constraint. Note that the propagation history used in ω_r serves to avoid the non-termination problem. In ω_{set}, **PropFire** performs actual propagation for the given matching to a propagation rule's head constraints, provided that matching conditions still hold. If the matching conditions do not hold, **PropDrop** prevents firing the rule's body constraints. The **Drop**$^<$ transition ensures that the constraint store is a set, and the same constraint is not activated over and over again.

3.2 Properties of ω_{set}

It is easy to show that ω_{set} is sound with respect to CHR's declarative semantics:

Theorem 1 (Soundness) *Let P be a CHR program, CT be the consistent theory underlying the built-in constraints in P, G be a goal, $\langle G, \emptyset, \mathit{true}\rangle \mapsto^*_{\mathit{set}} \langle E, C_U, C_B\rangle$ be a derivation, and C be the logical reading of the final state. Then $P, CT \models C \leftrightarrow G$.*

This theorem is established by induction on the length of a derivation.

Relationship between ω_{set} and ω_r. Since ω_{set} treats the constraint store as a set, programs for which ω_r places multiple occurrences of the same constraint in its store will have a different behavior under ω_{set} compared to ω_r. However, there are CHR programs for which the constraint store, even under ω_r, turns out to be a set. We call such programs set-CHR programs. Clearly, it is useful to compare the two semantics only for set-CHR programs.

In general, ω_{set} is not equivalent to ω_r. For instance, consider the evaluation of the CHR program in Fig. 2 in ω_{set} for the goal $\texttt{p}(\texttt{A}, \texttt{B})$. Starting from the empty constraint store, activation of $\texttt{p}(\texttt{A}, \texttt{B})$ will lead to store $\{p(A, B)\}$ (for brevity, we combine the user-defined and built-in stores in this example). Firing rule `r1` takes us to $\{p(A, B), q(A, B)\}$. Note that the simplification rule `r2` is not applicable in this store, but `r3` is, leading to the store $\{p(A, B), A = B\}$. Since variables A and B have new bindings in the store, the constraint $p(A, B)$ will be woken up by the **Solve** transition. Rule `r1` will be fired again, leading to the store $\{p(A, B), q(A, B), A = B\}$. Rule `r2` is applicable in this store, yielding $\{p(a, a)\}$. The evaluation terminates after one more round of **Solve** and firing of `r1` and `r2`.

```
r1 @ p(X,Y) ==> q(X,Y).
r2 @ q(X,X) <=> X = a.
r3 @ q(X,Y) <=> X = Y.
```

Fig. 2. CHR program with different fixed points in ω_r and ω_{set}

The evaluation in ω_r leads to a different derivation. In ω_r, each constraint is given an identifier to distinguish between different occurrences of the same constraint in the multi-set store. Propagation history is maintained in terms of the identifiers of matching constraints. When the variables in a constraint get bound, the constraint's identifier is not changed. This means that if a propagation rule was fired once for a set of matching constraints, it will not be fired again even when the variables in its matching constraints

are bound further. Thus evaluation of $p(A, B)$ in ω_r for the above example will proceed as in ω_{set} until we reach the store $\{p(A, B), A = B\}$. **Solve** will wake up $p(A, B)$, but rule `r1` will not be applicable since it was fired before for the same constraint. Hence, ω_r terminates with the store $\{p(A, B), A = B\}$!

It appears that the state with which ω_r terminates is not a fixed point, and the propagation history makes ω_r terminate the fixed point computation early. For instance, the evaluation of $p(A, B)$ terminates with a store equivalent to $p(A, A)$. But evaluation of $p(A, A)$ will terminate with a different store: $p(a, a)$! In contrast, ω_{set}'s termination condition (presence of a constraint in the store) distinguishes between a constraint term under different substitutions, and hence does not abandon the fixed point computation early. In general, there are set-CHR programs and goals that terminate with ω_r but not with ω_{set} due to this difference in identifying fixed points.

Datalog-CHR is a class of CHR programs such that (i) there are no function symbols of arity ≥ 1, and (ii) every variable in a rule occurs on the rule's left hand side. For instance, the program in Fig. 2 is a Datalog-CHR program. For programs in this class, evaluation using ω_{set} will terminate whenever ω_r terminates.

4 Compiling CHR with Distributed Constraint Store

Direct-Indexed CHR. We now define a subclass of CHR programs for which we can use a simple and efficient constraint store representation. Note that in ω_{set} **Simplify** and **PropMatch** select a matching substitution in order to determine whether the rule is applicable for a given active constraint. This operation significantly affects the efficiency of a CHR implementation, and a considerable amount of work has gone into devising index structures to optimize it [13, 23]. The matching procedure has two distinct parts: selecting from the store constraints that match the rule's head, and checking whether the guard is satisfiable under the matching substitution. The class of direct-indexed CHR programs, defined below, has a structure that simplifies the first part.

Each user-defined constraint in a direct-indexed program has a *mode* declaration that specifies the set of possible instances of the constraint that may appear in the store. Each argument of a constraint may have one of three modes: "$\mathtt{v}$" if that argument remains free in any instance of the constraint in the store, "$\mathtt{g}$" if that argument is a ground term all instances, and "$\mathtt{?}$" if that argument is a variable or a constant in all instances[3].

Given a constraint term c, we use $\mathit{avars}(c)$ to denote the set of variables that appear at positions with mode "$\mathtt{v}$". We assume that the mode declarations are consistent with the use of constraints in the rules and queries; and that all user-defined constraints have at least one position with mode "$\mathtt{v}$".

The *matching graph* for a (multi-)set of constraints is a graph in which there is a vertex representing each constraint in the set, and there is an edge between every pair of constraints that share a "$\mathtt{v}$"-moded variable. Formally,

Definition 1 *The matching graph of a set C of user-defined and built-in constraints is a labeled undirected graph $G = (V, E)$ where $V = C$, and E is the smallest set such that $\forall c_1, c_2 \in V, \mathit{avars}(c_1) \cap \mathit{avars}(c_2) \neq \emptyset \rightarrow (c_1, c_2, l) \in E$ where $l = \mathit{avars}(c_1) \cap \mathit{avars}(c_2)$.*

[3] Similar declarations have been used in other CHR systems [13, 23]

We can use the matching graph for a head of a CHR rule to drive the matching process. Given the vertex in the graph that matches the active constraint, we first can check whether its neighbors match any constraints in the store. Since a neighbor constraint shares unbound variables with the active constraint, we can index into the constraint store using this information, thereby speeding up matching. When the neighbors themselves are matched, we can traverse the graph further. Clearly, this process will not apply when there is a subset of head constraints that do not share variables with the remaining constraints in the head. The direct-indexed CHR is defined to disallow this condition. Formally:

Definition 2 *A rule R in a CHR program is said to be direct-indexed if the matching graph for its head constraints is connected. A CHR program is direct-indexed if all its rules are direct-indexed. A CHR goal is direct-indexed if its matching graph is connected. A CHR derivation is direct-indexed if it evaluates a direct-indexed goal over a direct-indexed program.*

All valid CHRd derivations are direct-indexed. Many CHR specifications, e.g., `leq` from Example 1, are naturally direct-indexed, and all CHR specifications can be trivially translated to direct-indexed CHR programs. We describe the issues surrounding such a translation at the end of this section.

The Distributed Constraint Store Representation. Following other CHR implementations, we use attributed variables [11] to represent constraints. Attributed variables are associated with mutable data, and an user-defined unification handler is invoked whenever an attributed variable is unified. In our implementation, a variable's attribute represents the set of all the constraints the variable participates in, that is, the variable's local constraint store. The attribute is encapsulated in a *constraint attribute term (CAT)*. The CAT is different from a *suspension term*, which in other Prolog implementations of CHR represents a single stored constraint. The CAT of a variable is a vector whose size is determined at compile time based on the number and arity of the user-defined constraint symbols. For instance, if `a/2` and `b/1` are the only two user-defined constraint symbols, and `a(X,Y)` is the lone constraint in the store, then `X`'s CAT will be `v([attr(Y)],[],[])`, and `Y`'s CAT will be `v([],[attr(X)],[])`.

The constraint store is a collection of constraint variables and their CATs. It should be noted that, although each argument of a CAT is represented as a list, it is manipulated as though it is a set. When two constrained variables are unified, their CATs will be merged. Again, we treat the arguments of the CATs as sets and compute their pairwise union. When a variable changes due to unification, all constraints in its CAT are considered to be in the *wakeup* set, and rules involving the constraints are re-fired.

The CHRd Compiler. Our compiler generates the code that faithfully implements the semantics ω_{set}, following the well-developed CHR-to-Prolog compilation schema [14]. In its current version, the compiler supports simple variants of the join-order, continuation, and late storage optimizations, standard in most of the traditional CHR systems.

Matching. CHRd's representation of the constraint store helps in quickly checking whether a matching constraint exists in the store. For instance, to select constraints of the form `a(U,Y)` for a particular variable `Y`, we need to simply inspect the *second* argument of `Y`'s CAT. This structure builds a single-level index on all arguments of a constraint. Although it is possible to build nested index structures within each argument of the CAT, this is not done in the current implementation of our system.

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=>
   N=<M | L is M-N, gcd(L).
```
(a)

```
:- mode gcd(v,g).
gcd(X,0) <=> true.
gcd(X,N) \ gcd(X,M) <=>
   N=<M | L is M-N, gcd(X,L).
```
(b)

Fig. 3. (a) A ground CHR program; and (b) its translation into direct-indexed CHR

During an application of a propagation rule, first the **PropMatch** transition retrieves all constraints of a desired form (by accessing appropriate arguments in the CATs of the constrained variables) into a temporary data structure. After all matchings for therule's head have been collected, the **PropFire** transition evaluates each matching in turn against the corresponding substitution of the rule's guard and, when the guard is satisfied, fires the rule's body constraints under the same substitution.

Evaluation of Ground CHR Programs. Consider the CHR program for evaluating the greatest common divisor of a set of integers given in Figure 3(a). When we pose two ground constraints, say `gcd(12)`, and `gcd(8)`, the program terminates with `gcd(4)` as the lone constraint in the store. The program is not direct-indexed since the matching graph for its second rule has two vertices and no edges (i.e. no shared variables).

Such programs can be trivially translated to direct-indexed CHR by adding an extra variable to each constraint. The direct-indexed CHR program equivalent to that in Figure 3(a) is given in Figure 3(b). The extra variable can be thought of as representing the constraint store itself. One salient point of the translation is that we now have a handle on a constraint store, and we can simultaneously create and manipulate multiple, possibly independent, stores. For instance, using the translated program, we can pose constraints `gcd(A,12), gcd(A,9)`, and in the same computation pose `gcd(B,45), gcd(B,30)`, and the two queries will be evaluated independently. Thus, we can consider the translated CHR program as operating over *local* constraint stores. The capacity for generating new constraint stores and manipulating them locally makes CHRd a good fit in a tabling system where each answer and call has an associated store.

5 Experimental Results

We now present the results of the experiments evaluating the performance of CHRd in tabled as well as non-tabled settings. All measurements were taken on a PC with 1.4 GHz Pentium-M processor and 512 MB RAM running Linux. The run time, given in milliseconds, is averaged over multiple tests. We have compared the performance of CHR and CHRd on XSB 3.0.1 (CHRd's native platform) and hProlog 2.4.35-32. We chose hProlog since it is the host for K.U.Leuven's CHR (KUL-CHR), currently the most representative of systems that efficiently implement Constraint Handling Rules.

Examples Using Tabled Evaluation We evaluate the performance of CHRd for tabled programs using four examples: (1) `truckload`, a problem used in [24] to measure the performance of XSB-CHR; (2) `buffer`, the constraint-based verification of "in-order" message delivery property of a FIFO buffer; (3) `dining_ph`, deadlock analysis of a dining philosophers specification using synchronization contracts; and (4) `fischer`, a CHR-based implementation of reachability analysis for real-time systems.

Benchmark	XSB-CHR	CHRd
truckload(300)	1870	243 (13%)
truckload(500)	2530	380 (15%)
fifo(240)	—	1580
fifo(320)	—	3730
dining_ph(6)	—	120
dining_ph(8)	—	980

Table 1. Run time (in ms.) for evaluation of tabled CHR programs

Truckload is a variant of *knapsack*, the classical dynamic programming problem, for scheduling the delivery of packages using finite-capacity trucks to different destinations. Tabling ensures polynomial-time behavior. The base data for the problem, e.g. the attributes of packages were taken from [24].

The time taken to run truckload in XSB-CHR and CHRd for two truck capacities is given in Table 1. The percentage of time taken by CHRd w.r.t. XSB-CHR, given in parentheses, shows that CHRd is four times faster than XSB-CHR. The memory usage is similar on both systems.

The truckload problem is relatively small, and CHRd significantly outperformed XSB-CHR. The other tabled problems, taken from verification examples, are relatively large. As can be seen from the table, due to the more complex constraints and large number of table operations, XSB-CHR failed to work on these examples.

In [20] we presented a constraint-based algorithm for verifying a class of infinite-state systems called *data independent systems*. The rows labeled fifo(N) of Table 1 show the run time of a CHRd-based implementation of this algorithm for verifying the "in-order" message delivery property of an N-place FIFO buffer. The solver uses a reachability-based algorithm and hence needs tabling for termination. The original implementation for this problem (which used a meta-interpreter for constraint handling) is 2.5 times slower that the one based on CHRd.

Our deadlock detection framework [21] uses CHR to enforce correct synchronization of threads based on locally defined concurrency constraints, and reachability analysis to detect deadlocked states. Table 1 shows run time results for the evaluation of two configurations of N dining philosophers in which no deadlock was found.

Finally, we used CHRd to analyze *Fischer's protocol*, a mutual-exclusion protocol that is often used to benchmark real-time verification tools. We used CHR to specify a solver for the clock constraints. While CHRd was able to solve the verification problems for various instances of the protocol, XSB-CHR was unable to solve even a 2-process instance. However, the CHRd-based verifier is 2-5 times slower than a verifier that uses a hand-built (Prolog-based) clock constraint solver [17]. This example indicates that CHRd needs to be further optimized before it can compete with custom-built solvers for well-known constraint domains.

Non-tabled Examples Table 2 compares CHRd running on XSB and hProlog, with each platform's original CHR system: XSB-CHR and KUL-CHR. The table shows the results for direct-indexed programs: cycle, a cycle of leq constraints on N variables; queens and zebra, two classical problems solved using finite-domain CHR; bool, N-digit binary addition; bool_chain, a cycle of "$\wedge$" constraints over N variables; alias, an encoding of Anderson's may-points-to analysis for C programs [2]; and ta, an evaluation of clock bounds on finite automata.

The table also shows results for ground CHR benchmarks: gcd described in Section 4; primes, a computation of prime numbers up to N; fib, a computation of first N Fibonacci numbers; and ram_simul, a simulator of a RAM machine. The ground

Benchmark	XSB		hProlog	
	XSB-CHR	CHRd	KUL-CHR	CHRd
`cycle(60)`	11015	1500 (14%)	940	554 (59%)
`queens(16)`	9693	2520 (26%)	1250	820 (65%)
`zebra(10)`	45220	690 (2%)	1130	320 (28%)
`bool(50000)`	255810	1470 (1%)	770	1050 (136%)
`bool_chain(400)`	207420	16970 (8%)	680	610 (90%)
`alias(m88ksim)`	–	189	140	96 (69%)
`alias(parser)`	–	3690	3650	4050 (111%)
`ta(200)`	5860	2090 (35%)	690	560 (75%)
`gcd([3,10^6])`	1010	945 (94%)	126	360 (300%)
`primes(2000)`	6871	2753 (40%)	500	1065 (213%)
`fib(500)`	3260	1250 (38%)	240	475 (198%)
`ram_simul(40000)`	–	2740	330	1170 (355%)

Table 2. Runtime (in ms.) for evaluation of non-tabled CHR programs

CHR programs were evaluated directly by the native CHR systems on each platform. For CHRd, they were first translated as described in Section 4 and then ran.

Clearly, CHRd outperforms XSB-CHR for all tests. On hProlog, the performance of CHRd is close to, or better than, that of KUL-CHR for the direct-indexed programs. It should be noted that KUL-CHR is built to handle a more general class of CHR programs, and does not exploit the indexing available in direct-indexed programs. On the other hand, CHRd does not (currently) optimize the compilation based on the mode information, nor does it support the alternative index structures (e.g. hash tables) used in KUL-CHR. The significantly slower run times of our system for the ground benchmarks is due to the absence of such optimizations. We believe that adding these optimizations to CHRd will bring its performance closer to that of KUL-CHR.

6 Related Work and Discussion

Although the CHR framework was initially proposed for specifying constraint solvers, there is a growing body of work for using it as a full-fledged programming language. The semantics of the language and its implementation have evolved hand-in-hand. For instance, while the initial papers refer to the constraint store as a conjunction of constraints [10], the implementations represented the store using multi-set of terms. Subsequently, the formalization of its semantics in terms of multi-set rewriting have been widely accepted. The original operational semantics [1, 10] has been refined [9] to reduce non-determinism and extend the class of programs amenable for evaluation. One of the stated motivations for the refined semantics was to bring the formalism closer to the popular implementations.

It was observed that the propagation history, a key structure of CHR semantics, contributed to significant performance issues when an existing CHR implementation was ported to a tabling environment [24]. When working with multi-set-based constraint store, it appears that propagation history is essential to provide a reasonable semantics. Our work can be viewed as an investigation into the effect of making the constraint

store set-based. Note that bottom-up techniques for evaluating definite logic programs compute fixed points (minimal models) without maintaining something analogous to propagation history [16]. Our semantics ω_{set} extends this basic idea to work in the presence of simplification rules (i.e. non-monotonic changes to the store) and bindings on variables. As a result, we obtain a simpler semantics that is easier to implement in a tabled setting. We ensure that the new semantics is as close as possible to existing implementations by basing its formulation on refined semantics ω_r [9]. Although our semantics ω_{set} coincides with ω_r for a large class of constraint handlers written in CHR, the two semantics do not coincide in general. One problem of current interest is to identify the class of CHR programs for which the two semantics coincide.

Although traditional CHR implementations rely on central storage of constraints, direct indexing (storing constraints as variable attributes) has been recognized as more efficient. Therefore, many existing systems [13, 14, 23] store constraints both ways, using the central data structures only to access constraints when direct indexing is not available. Our work extends this approach by entirely eliminating central storage, and transforming programs that are not direct-indexed to simulate a store using another attributed variable. All CHR implementations we are aware of maintain a propagation history, which is eliminated in CHRd, thanks to its set-based semantics. Consequently, we have reduced the overheads of storing and manipulating constraint stores, leading to a scalable integration of CHR-based constraint solvers with tabled evaluation.

As mentioned before, CHR is being treated as a full-fledged programming language, and not just for writing constraint solvers. It has been shown that algorithms can be encoded in CHR and evaluated with no loss in their asymptotic-time complexity [25]. Recent works have addressed the space complexity of CHR programs [26]. In order to support the growing number of applications (most of them are ground CHR programs), a lot of effort has gone to optimizing central storage structures critical to performance of such programs. Works such as [13] propose analyses to determine the best index depending on the properties of the constraints specified in the program. Additionally, structures that guarantee efficient lookup (234-trees in [13] or hash tables in [23]) have replaced simple unordered lists that was used in early implementations [14]. In CHRd, the constraint set associated with each variable is defined as an unordered list, similar to that in [14]. Incorporating the results of indexing research will improve the CHRd implementation. Furthermore, for non-tabled programs, CHRd replaces the check on propagation history by a check on the constraint store. Our experience with CHRd indicates that constraint store checks can be done as efficiently as propagation history checks. There has also been analyses that determine whether propagation history can be eliminated [22]. Whether similar analyses can be used to eliminate explicit checks on the constraint store remains to be seen.

References

1. S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *CP '97*, pages 252–266, 1997.
2. L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Unversity of Copenhagen, 1994.
3. M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–154, 2004.

4. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
5. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *ACM PLDI*, 1996.
6. G. Delzanno and T. Bultan. Constraint-based verification of client-server protocols. In *CP*, pages 286–301, 2001.
7. G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS*, pages 223–239, 1999.
8. B. Demoen. hProlog. `http://www.cs.kuleuven.ac.be/~bmd/hProlog/`.
9. G. J. Duck, P. J. Stuckey, M. J. G. de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP 2004*, pages 90–104, 2004.
10. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.
11. C. Holzbaur. Metastructures versus Attributed Variables in the Context of Extensible Unification. In *PLILP '92*, pages 260–268, 1992.
12. C. Holzbaur, M. G. de la Banda, D. Jeffery, and P. J. Stuckey. Optimizing Compilation of Constraint Handling Rules. In *ICLP 2001*, volume 2237 of *Lecture Notes in Computer Science*, 2001.
13. C. Holzbaur, M. G. de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of constraint handling rules in HAL. *Theory and Practice of Logic Programming*, 5(4-5, Special Issue on Constraint Handling Rules):503–531, 2005.
14. C. Holzbaur and T. W. Frühwirth. Compiling Constraint Handling Rules into Prolog with Attributed Variables. In *PPDP '99*, pages 117–133, 1999.
15. N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *PADL*, pages 58–73, 2003.
16. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.
17. G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient model checking of real time systems using tabled logic programming and constraints. In *ICLP*, 2002.
18. C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *CAV*, volume 1855 of *LNCS*, pages 576–580, 2000.
19. K. Sagonas, T. Swift, D. S. Warren, P. Rao, and J. Friere. The XSB logic programming system. `http://xsb.sourceforge.net`.
20. B. Sarna-Starosta and C. R. Ramakrishnan. Constraint-based model checking of data-independent systems. In *Intl. Conf. on Formal Engineering Methods (ICFEM)*, volume 2885 of *LNCS*, pages 579–598, 2003.
21. B. Sarna-Starosta, R. E. K. Stirewalt, and L. K. Dillon. A model-based design-for-verification approach to checking for deadlock in multi-threaded systems. In *18th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2006.
22. T. Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, 2005.
23. 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. Published as technical report: Ulmer Informatik-Berichte Nr. 2004-01.
24. T. Schrijvers and D. S. Warren. Constraint handling rules and tabled execution. In *ICLP*, pages 120–136, 2004.
25. J. Sneyers, T. Schrijvers, and B. Demoen. The Computational Power and Complexity of Constraint Handling Rules. In *CHR 2005*, 2005.
26. J. Sneyers, T. Schrijvers, and B. Demoen. Memory reuse for CHR. In *ICLP 2006*, 2006.
27. Swedish Institute of Computer Science. SICStus Prolog System. `http://www.sics.se/isl/sicstuswww/site/index.html`.
28. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *ICLP*, pages 84–98, 1986.