As Time Goes By: Constraint Handling Rules

A Survey of CHR Research from 1998 to 2007

JON SNEYERS, PETER VAN WEERT,

TOM SCHRIJVERS, and LESLIE DE KONINCK

Dept. of Computer Science, K.U.Leuven, Belgium

(*e-mail:* `FirstName.LastName@cs.kuleuven.be`)

Abstract

Constraint Handling Rules (CHR) is a high-level programming language based on multi-headed multiset rewrite rules. Originally designed for writing user-defined constraint solvers, it is now recognized as an elegant general purpose language.

CHR-related research has surged during the decade following the previous survey by Frühwirth (1998). Covering more than 180 publications, this new survey provides an overview of recent results in a wide range of research areas, from semantics and analysis to systems, extensions and applications.

KEYWORDS: Constraint Handling Rules, CHR, survey

Contents

1 Introduction

Constraint Handling Rules (CHR) is a high-level programming language based on multi-headed, committed-choice, guarded multiset rewrite rules. Originally designed in 1991 by Frühwirth (1992; 1995; 1998; 2009) for the special purpose of adding user-defined constraint solvers to a host-language, CHR has matured over the last decade to a powerful and elegant general-purpose language with a wide spectrum of application domains. Its logical semantics and monotonicity properties naturally lead to anytime, online, and concurrent programs.

The previous survey on CHR (Frühwirth 1998) was written in 1998. The aim of this paper is to complement that survey by giving an overview of the last decade of CHR-related research. We advise readers that are not yet familiar with CHR to read Frühwirth (1998) first as we have kept the amount of overlap minimal.

Overview. We start with a short historical overview of the past 10 years of CHR research, followed by an introduction to the language itself. Section 2 describes the logical and operational semantics; Section 3 covers program analysis topics such as confluence, termination, and complexity. Next, in Section 4, we discuss the different CHR systems and compilation techniques. Extensions and variants of CHR are dealt with in Section 5, while Section 6 discusses the relation between CHR and other formalisms. In Section 7 we give an overview of the many applications of CHR. Finally, Section 8 concludes this survey.

1.1 Historical Overview

Early CHR research is performed at the Ludwig Maximilians Universität (LMU) and the European Computer-Industry Research Centre (ECRC), both in Munich, by Frühwirth (who later moves to Ulm) and his students Abdennadher (who later moves to Cairo), Meuss, and Wolf (in Berlin).

At the end of the nineties, CHR research focusses on theoretical properties of
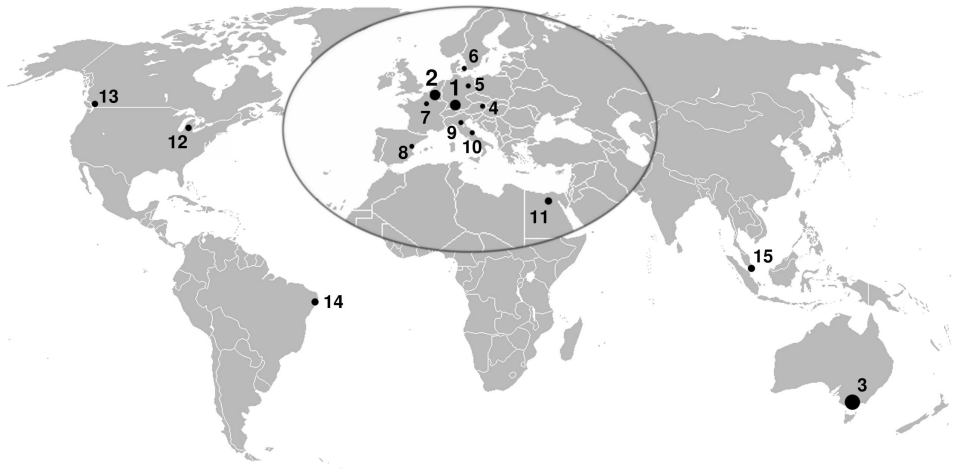
Fig. 1. CHR research groups all over the world: 1. Ulm, Germany (Frühwirth, Meister, Betz, Djelloul, Raiser, . . .); 2. Leuven, Belgium (Schrijvers, Demoen, Sneyers, De Koninck, Van Weert, . . .); 3. Melbourne, Australia (Duck, Stuckey, García de la Banda, Wazny, Brand, . . .); 4. Vienna, Austria (Holzbaur); 5. Berlin, Germany (Wolf); 6. Roskilde, Denmark (Christiansen); 7. Paris, France (Coquery, Fages); 8. Castellón, Spain (Escrig, . . .); 9. Bologna/Ferrara, Italy (Mello, Lamma, Tacchella, . . .); 10. Chieti, Italy (Meo, . . .); 11. Cairo, Egypt (Abdennadher, . . .); 12. Michigan, USA (Sarna-Starosta); 13. Vancouver, Canada (Dahl); 14. Recife, Brazil (Robin, Vitorino, . . .); 15. Singapore (Sulzmann, Lam);

CHR programs like confluence, completion, operational equivalence, termination, and complexity (Section 3). In the same period, the seminal Holzbaur-Frühwirth CHR system in SICStus Prolog is developed (Holzbaur and Frühwirth 1998; 1999; 2000a) and the first CHR systems in Java are created (Section 4.1.3).

Until about 2001, most of the CHR research is still done in Germany and Vienna; other groups are discovering CHR, at first mostly as an implementation language for applications. For instance, Sulzmann et al. use CHR for type systems (Section 7.3.1) and Christiansen and Dahl develop CHR grammars for language processing (Section 7.3.3). Meanwhile, Brand, Monfroy, Abdennadher and Rigotti study the automatic generation of CHR programs (Section 7.1.5). Starting around 2002 there is a strong growth of international research interest in CHR, leading to a series of workshops on CHR (Frühwirth and Meister 2004; Schrijvers and Frühwirth 2005b; Schrijvers and Frühwirth 2006; Djelloul, Duck et al. 2007). Figure 1 gives an (incomplete) overview of the most active CHR research groups all over the world.

In 2003 and 2004, the groups in Melbourne and Leuven start working on (static) analysis and optimizing compilation of CHR, culminating in the Ph.D. theses of Duck (2005) and Schrijvers (2005). This work leads to the formulation of the refined operational semantics (Section 2.2.2) and the creation of new, highly optimizing CHR systems (Section 4.2).

In Leuven and Brazil, research starts around 2005 on search and Java implementations of CHR (Sections 4.1.3 and 5.2.1), while the Ulm group investigates alternative logical semantics for CHR (Section 2.1). The study of an implementa-

tion of the union-find algorithm in CHR leads to a focus on compiler optimizations (Section 4.2.2) and the use of CHR for general-purpose programming (Section 7.2).

Recent trends in CHR-related research include the relation between CHR and other formalisms (Section 6), extensions and variants of CHR (Section 5), and a renewed interest in theoretical properties like confluence, termination, and complexity (Section 3). In the context of the NICTA project "G12", the Melbourne group is currently developing Cadmium, an ACD term rewriting language which extends CHR (Section 6). The Ulm group is currently researching global constraints in the context of the DFG project "GLOBCON"; this work is related to automatic rule generation (Section 7.1.5) and program transformation (Section 4.2.3).

1.2 Constraint Handling Rules

To make this survey somewhat self-contained, we briefly introduce the syntax and informal semantics of Constraint Handling Rules. For a gentler introduction to CHR, we refer the reader to Frühwirth (1998), Frühwirth and Abdennadher (2003), Schrijvers (2005), Duck (2005), or Frühwirth (2009).

CHR is embedded in a host language $\mathcal{H}$ that provides data types and a number of predefined constraints. These constraints are called host language constraints or *built-in* constraints. The traditional host language of CHR is Prolog. Its only host language constraint is equality of Herbrand terms; its data types are Prolog variables and terms. We denote the host language in which CHR is embedded between round brackets: i.e. CHR($\mathcal{H}$) denotes CHR embedded in host language $\mathcal{H}$. Most systems are CHR(Prolog) systems, but there are also several implementations of CHR(Java) and CHR(Haskell), and recently a CHR(C) system was developed. A thorough discussion of existing CHR implementations is given in Section 4. We require the host language to provide at least the basic constraints *true* and *fail*, and syntactic equality ("==") and inequality ("\==") checks.

1.2.1 Syntax

CHR constraint symbols are drawn from the set of predicate symbols, denoted by a functor/arity pair. CHR constraints, also called constraint atoms or constraints for short, are atoms constructed from these symbols and the data types provided by the host language. A CHR program $\mathcal{P}$ consists of a sequence of CHR rules. There are three kinds of rules: (where $l, m, n, o \geq 1$)

- Simplification rules: $\qquad\qquad\qquad h_1, \ldots, h_n \iff g_1, \ldots, g_m \mid b_1, \ldots, b_o.$
- Propagation rules: $\qquad\qquad\qquad h_1, \ldots, h_n \implies g_1, \ldots, g_m \mid b_1, \ldots, b_o.$
- Simpagation rules: $\quad h_1, \ldots, h_l \setminus h_{l+1}, \ldots, h_n \iff g_1, \ldots, g_m \mid b_1, \ldots, b_o.$

The sequence, or conjunction, $h_1, \ldots, h_n$ are CHR constraints; together they are called the *head* or *head constraints* of the rule. A rule with n head constraints is called an *n-headed* rule and when $n > 1$, it is a *multi-headed* rule. All the head constraints of a simplification rule and the head constraints $h_{l+1}, \ldots, h_n$ of a simpagation rule are called *removed* head constraints. The other head constraints — all

4

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

Fig. 2. The CHR(Prolog) program LEQ, a solver for the less-than-or-equal constraint.

heads of a propagation rule and $h_1, \ldots, h_l$ of a simpagation rule — are called *kept* head constraints. The conjunction $b_1, \ldots, b_o$ consists of CHR constraints and host language constraints; it is called the *body* of the rule. The part of the rule between the arrow and the body is called the *guard*. It is a conjunction of host language constraints. The guard "$g_1, \ldots, g_m \mid$" is optional; if omitted, it is considered to be "*true* $\mid$". A rule is optionally preceded by *name* @ where *name* is a term. No two rules may have the same name, and rules without an explicit name get a unique name implicitly.

For simplicity, both simplification and propagation rules are often treated as special cases of simpagation rules. The following notation is used:

$$H_k \setminus H_r \iff G \mid B$$

If H_k is empty, then the rule is a simplification rule. If H_r is empty, then the rule is a propagation rule. At least one of H_r and H_k must be non-empty.

1.2.2 Informal Semantics

A derivation starts from an initial query: a multiset of constraint atoms, given by the user. This multiset of constraints is called the *constraint store*. The derivation proceeds by applying the rules of the program, which modify the constraint store. When no more rules can be applied, the derivation ends; the final constraint store is called the solution or solved form.

Rules modify the constraint store in the following way. A simplification rule can be considered as a rewrite rule which replaces the left-hand side (the head constraints) with the right-hand side (the body constraints), on the condition that the guard holds. The double arrow indicates that the head is logically equivalent to the body, which justifies the replacement. The intention is that the body is a simpler, or more canonical form of the head.

In propagation rules, the body is a consequence of the head: given the head, the body may be added (if the guard holds). Logically, the body is implied by the head so it is redundant. However, adding redundant constraints may allow simplifications later on. Simpagation rules are a hybrid between simplification rules and propagation rules: the constraints before the backslash are kept, while the constraints after the backslash are removed.

1.2.3 Examples

The program LEQ (Fig. 2) is a classic example CHR program to solve less-than-or-equal constraints. The first rule, *reflexivity*, replaces the trivial constraint **leq**(X,X)

generate	@ **upto**(N) $\iff$ N > 1 | **prime**(N), **upto**(N-1).
done	@ **upto**(1) $\iff$ *true*.
remove_nonprime @	**prime**(A) \ **prime**(B) $\iff$ B mod A = 0 | *true*.

Fig. 3. The CHR program PRIMES, a prime number sieve.

by *true*. Operationally, this entails removing this constraint from the constraint store (the multiset of all known CHR constraints). The second rule, *antisymmetry*, states that **leq**(X,Y) and **leq**(Y,X) are logically equivalent to X = Y. Operationally this means that constraints matching the left-hand side may be removed from the store, after which the Prolog built-in equality constraint solver is used to unify X and Y. The third rule, *idempotence*, removes redundant copies of the same **leq**/2 constraint. It is necessary to do this explicitly since CHR has multiset semantics. The last rule, *transitivity*, is a propagation rule that computes the transitive closure of the **leq**/2 relation. An example derivation could be as follows:

		leq(A,B), **leq**(B,C), **leq**(C,A)
(transitivity)	$\rightarrowtail$	**leq**(A,B), **leq**(B,C), **leq**(C,A), **leq**(B,A)
(antisymmetry)	$\rightarrowtail$	**leq**(B,C), **leq**(C,A), A = B
(Prolog)	$\rightarrowtail$	**leq**(A,C), **leq**(C,A), A = B
(antisymmetry)	$\rightarrowtail$	A = C, A = B

Figure 3 lists another simple CHR(Prolog) program called PRIMES, a CHR variant of the Sieve of Eratosthenes. Dating back to at least 1992 (Frühwirth 1992), this is one of the very first examples where CHR is used as a general-purpose programming language. Given a query of the form "**upto**(n)", where n is a positive integer, it computes all prime numbers up to n. The first rule (*generate*) does the following: if $n > 1$, it 'simplifies' **upto**(n) to **upto**($n-1$) and adds a **prime**(n) constraint. The second rule handles the case for $n = 1$, removing the **upto**(1) constraint. Note that removing a constraint is done by simplifying it to the built-in constraint *true*. The third and most interesting rule (*remove_nonprime*) is a simpagation rule. If there are two **prime**/1 constraints **prime**(A) and **prime**(B), such that B is a multiple of A, the latter constraint is removed. The effect of the *remove_nonprime* rule is to remove non-primes. As a result, if the rules are applied exhaustively, the remaining constraints correspond exactly to the prime numbers up to n.

2 Semantics

In this section, we give an overview of both the logical (declarative) semantics and the operational semantics of CHR. The logical semantics (Section 2.1) constitute the formal foundations for the CHR programming language, whilst the operational semantics (Section 2.2) determine the behavior of actual implementations.

6

2.1 Logical Semantics

2.1.1 Classical Logic Semantics

Let $\bar{x}$ denote the variables occurring only in the body of the rule. We use $\bar{\forall}(F)$ to denote universal quantification over all free variables in F. A simplification rule $H \iff G \mid B$ corresponds to a logical equivalence, under the condition that the guard is satisfied: $\bar{\forall}(G \to (H \leftrightarrow \exists \bar{x} B))$. A propagation rule $H \implies G \mid B$ corresponds to a logical implication if the guard is satisfied: $\bar{\forall}(G \to (H \to \exists \bar{x} B))$. A simpagation rule $H_k \setminus H_r \iff G \mid B$ corresponds to a conditional equivalence: $\bar{\forall}(G \to (H_k \to (H_r \leftrightarrow \exists \bar{x} B)))$. The (classical) logical semantics (Frühwirth 1998) of a CHR program — also called its logical reading, declarative semantics, or declarative interpretation — is given by the built-in constraint theory $\mathcal{D}_{\mathcal{H}}$ (which defines the built-ins of the host language $\mathcal{H}$) in conjunction with the logical formulas for each rule. As an example, consider the program LEQ of Fig. 2. The logical formulas corresponding to its rules are the following:

$$\left\{ \begin{array}{ll} \forall x, y : x = y \to (\mathbf{leq}(x, y) \leftrightarrow true) & \textit{(reflexivity)} \\ \forall x, y, x', y' : x = x' \land y = y' \to (\mathbf{leq}(x, y) \land \mathbf{leq}(y', x') \leftrightarrow x = y) & \textit{(antisymmetry)} \\ \forall x, y, x', y' : x = x' \land y = y' \to (\mathbf{leq}(x, y) \to (\mathbf{leq}(x', y') \leftrightarrow true)) & \textit{(idempotence)} \\ \forall x, y, y', z : y = y' \to (\mathbf{leq}(x, y) \land \mathbf{leq}(y', z) \to \mathbf{leq}(x, z)) & \textit{(transitivity)} \end{array} \right.$$

or equivalently:

$$\left\{ \begin{array}{ll} \forall x : \mathbf{leq}(x, x) & \textit{(reflexivity)} \\ \forall x, y : \mathbf{leq}(x, y) \land \mathbf{leq}(y, x) \leftrightarrow x = y & \textit{(antisymmetry)} \\ true & \textit{(idempotence)} \\ \forall x, y, z : \mathbf{leq}(x, y) \land \mathbf{leq}(y, z) \to \mathbf{leq}(x, z) & \textit{(transitivity)} \end{array} \right.$$

Note the strong correspondence between the syntax of the CHR rules, their logical reading, and the natural definition of partial order.

The classical logical reading, however, does not reflect CHR's multiset semantics (the *idempotence* rule is logically equivalent to *true*). Also, the classical logic reading does not always make sense. For example, consider the classical logic reading of the PRIMES program of Fig. 3:

$$\left\{ \begin{array}{ll} \forall n : n > 1 \to \mathbf{upto}(n) \leftrightarrow \exists n' \mathbf{prime}(n) \land n' = n - 1 \land \mathbf{upto}(n') & \textit{(generate)} \\ \mathbf{upto}(1) \leftrightarrow true & \textit{(done)} \\ \forall a, b : a | b \to \mathbf{prime}(a) \to (\mathbf{prime}(b) \leftrightarrow true) & \textit{(remove\_nonprime)} \end{array} \right.$$

which is equivalent to:

$$\left\{ \begin{array}{ll} \forall n > 1 : \mathbf{upto}(n) \leftrightarrow \mathbf{prime}(n) \land \mathbf{upto}(n - 1) & \textit{(generate)} \\ \mathbf{upto}(1) & \textit{(done)} \\ \forall a, b : \mathbf{prime}(a) \land a | b \to \mathbf{prime}(b) & \textit{(remove\_nonprime)} \end{array} \right.$$

The last formula nonsensically states that a number is prime if it has a prime factor.

2.1.2 Linear Logic Semantics

For general-purpose CHR programs such as PRIMES, or programs that rely on CHR's multiset semantics, the classical logic reading is often inconsistent with the intended meaning (see previous section). To overcome these limitations, Bouissou (2004) and Betz and Frühwirth (2005; 2007) independently proposed an alternative declarative semantics based on (intuitionistic) linear logic. The latter, most comprehensive study provides strong soundness and completeness results, as well as a semantics for the CHR$^\vee$ extension of CHR (see Section 5.2.1). For CHR programs whose constraints represent a multiset of resources, or whose rules represent unidirectional actions or updates, a linear logic semantics proves much more appropriate. A simple example is the following coin-throwing simulator (which depends on the nondeterminism in the operational semantics):

$$\textbf{throw}(\text{Coin}) \iff \text{Coin} = \texttt{head}.$$
$$\textbf{throw}(\text{Coin}) \iff \text{Coin} = \texttt{tail}.$$

The classical logic reading of this program entails $\texttt{head} = \texttt{tail}$. The linear logic reading of the coin-throwing program boils down to the following formula:

$$!(\textbf{throw}(Coin) \multimap (Coin = \texttt{head}) \& (Coin = \texttt{tail}))$$

In natural language, this formula means "you can always replace $\textbf{throw}(Coin)$ with either $(Coin = \texttt{head})$ or $(Coin = \texttt{tail})$, but not both". This corresponds to the committed-choice and unidirectional rule application of CHR.

2.1.3 Transaction Logic Semantics

The linear logic semantics is already closer to the operational semantics than the CHR classical logical semantics. However, it still does not allow precise reasoning about CHR derivations: while derivations correspond to proofs of logic equivalence of the initial and the final state, it only allows reasoning on the result of an execution, not on the execution itself. The transaction logic semantics (Meister, Djelloul et al. 2007) bridges the remaining gap between the logical and operational semantics of CHR by providing a framework for both inside one formal system.

2.2 Operational Semantics

The behavior of CHR implementations is determined by their operational semantics. As the original theoretical semantics of CHR (Section 2.2.1) proved too nondeterministic for practical programming, more deterministic instances have been specified that offer more execution control (Sections 2.2.2 and 2.2.3).

2.2.1 Theoretical Operational Semantics ω_t

The operational semantics ω_t of CHR (Frühwirth 1998), sometimes also called *theoretical* or *high-level* operational semantics, is highly nondeterministic. It is formulated as a state transition system.

Definition 2.1
An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i, the *constraint identifier*. This number serves to differentiate between copies of the same constraint. We introduce the functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets of identified CHR constraints in the obvious manner, e.g., $id(S) = \{i|c\#i \in S\}$.

Definition 2.2
An *execution state* σ is a tuple $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$. The *goal* $\mathbb{G}$ is a multiset of constraints to be rewritten to solved form. The CHR constraint *store* $\mathbb{S}$ is a set of *identified* CHR constraints that can be matched with rules in the program $\mathcal{P}$. Note that $chr(\mathbb{S})$ is a multiset although $\mathbb{S}$ is a set. The *built-in constraint store* $\mathbb{B}$ is the conjunction of all built-in constraints that have been posted to the underlying solver. These constraints are assumed to be solved (implicitly) by the host language $\mathcal{H}$. The *propagation history* $\mathbb{T}$ is a set of tuples, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. The propagation history is used to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints have not been used to fire the same rule before.[1] Finally, the counter $n \in \mathbb{N}$ represents the next integer that can be used to number a CHR constraint. We use $\sigma, \sigma_0, \sigma_1, \ldots$ to denote execution states and Σ^{CHR} to denote the set of all execution states.

For a given CHR program $\mathcal{P}$, the transitions are defined by the binary relation $\rightarrowtail_{\mathcal{P}} \subset \Sigma^{\text{CHR}} \times \Sigma^{\text{CHR}}$ shown in Figure 4. Execution proceeds by exhaustively applying the transition rules, starting from an initial state.

The **Solve** transition solves a built-in constraint from the goal, the **Introduce** transition inserts a new CHR constraint from the goal into the CHR constraint store, and the **Apply** transition fires a rule instance. A rule instance instantiates a rule with CHR constraints matching the heads, using a matching substitution (a one-way variable substitution).

Relatively strong soundness and completeness results (Frühwirth 1998) link the logical semantics and the operational semantics. Maher (2002) discusses the notion of *propagation completeness* and proves an impossibility result for CHR.

Variants of ω_t have been introduced to formalize extensions and variants of CHR. For example, the operational semantics of CHR$^{\vee}$(Abdennadher 2000), probabilistic CHR (Frühwirth, Di Pierro et al. 2002), and CHR with aggregates (Sneyers, Van Weert et al. 2007) are all based on ω_t. We discuss these and other extensions in Section 5.

---

[1] Early work on CHR, as well as some more recent publications (e.g., Bouissou 2004; Duck, Stuckey et al. 2007; Haemmerlé and Fages 2007), use a *token store* instead of a propagation history (this explains the convention of denoting the propagation history with $\mathbb{T}$). A token store contains a token for every potential (future) propagation rule application, which is removed when the rule is actually applied. The propagation history formulation is dual, but closer to most implementations. Confusingly, the term *token store* has also been used for what is commonly referred to as the "propagation history" (e.g., Chin, Sulzmann et al. 2003; Tacchella, Gabbrielli et al. 2007).

Fig. 4. The transition rules of the theoretical operational semantics ω_t, defining $\rightarrowtail_{\mathcal{P}}$. We use $\uplus$ for multiset union. For constraint conjunctions B_1 and B_2, $\bar{\exists}_{B_2}(B_1)$ denotes $\exists X_1, \dots, X_n : B_1$, with $\{X_1, \dots, X_n\} = vars(B_1) \setminus vars(B_2)$.

We should also mention the work on an and-compositional semantics for CHR (Delzanno, Gabbrielli et al. 2005; Gabbrielli and Meo 2009), which allows one to retrieve the semantics of a conjunctive query given the semantics of the conjuncts. This property is a first step towards incremental and modular analysis and verification tools.

2.2.2 Refined Operational Semantics ω_r

The refined operational semantics ω_r (Duck, Stuckey et al. 2004) instantiates the ω_t operational semantics by removing much of the nondeterminism. It formally captures the behavior of many CHR implementations (see also Section 4). CHR programs often rely on the execution control offered by the ω_r semantics for correctness, or to achieve a good time complexity.

The refined operational semantics uses a stack of constraints: when a new constraint arrives in the constraint store it is pushed on the stack. The constraint on top of the stack is called the *active* constraint. The active constraint attempts to match rule heads, together with suitable constraints from the constraint store (*partner* constraints), All occurrences of the active constraint are tried in the order in which they occur in the program. When all occurrences have been tried, the constraint is popped from the stack. When a rule fires, its body is executed immediately from left to right, thereby potentially suspending the active constraint because of newly arriving constraints. When a constraint becomes topmost again, it resumes its search for matching clauses.

Alternative formalizations of the ω_r semantics have been made for easier reasoning about certain optimizations or analyses. Examples are the call-based refined operational semantics ω_c (Schrijvers, Stuckey et al. 2005) and the semantics for occurrence representations ω_o (Sneyers, Schrijvers et al. 2005). Variants of ω_r have also been introduced to formalize implementations of proposed extensions to CHR or variants of CHR. To mention just a few of them: the $\omega_r^{\neg}$ semantics for CHR$^{\neg}$ (Van Weert, Sneyers et al. 2006), the $\omega_r^{\vee}$ semantics for CHR$^{\vee}$ which is equivalent to the tree-based ω_{λ} semantics (De Koninck, Schrijvers et al. 2006b), the set-based ω_{set} semantics of CHRd (Sarna-Starosta and Ramakrishnan 2007), and the concurrent refined semantics (Lam and Sulzmann 2007). These extensions and variants are discussed in more detail in Section 5.

2.2.3 Priority Semantics ω_p

While the refined operational semantics reduces most of the nondeterminism of the ω_t semantics, it arguably does not offer the CHR programmer an intuitive and predictable way to influence control flow. The ω_r semantics in a sense forces the programmer to understand and take into account how CHR implementations work, to achieve the desired execution control. De Koninck, Schrijvers et al. (2007b) introduced the extension CHR$^{\mathrm{rp}}$, with a corresponding operational semantics called ω_p. The programmer assigns a priority to every rule. The ω_p semantics is an instantiation of ω_t which ensures that of all applicable rules, the one with the highest priority is applied first. This feature gives the programmer a much more precise and high-level control over program execution compared to the ω_r semantics.

3 Program Analysis

In this section we discuss important properties of CHR programs: confluence (Section 3.1), termination (Section 3.2), and complexity (Section 3.3), as well as (semi-) automatic analysis of these properties. Program analyses that are mostly used for optimizing compilation are discussed in Section 4.2.2.

3.1 Confluence

If for a given CHR program, for all initial states, any ω_t derivation from that state results in the same final state, the program is called *confluent*. Confluence has been investigated thoroughly in the context of CHR (Abdennadher, Frühwirth et al. 1999). Two important results are discussed already in (Frühwirth 1998): the existence of a decidable, sufficient and necessary test for confluence of terminating programs, and the result that confluence implies correctness (consistency of the logical reading). Confluence under the refined ω_r semantics is investigated in Chapter 6 of (Duck 2005), which also discusses a refined confluence test.

Recently, the topic of confluence received renewed attention because certain problems and limitations of the confluence test have surfaced. Firstly, many programs that are in practice confluent fail this confluence test because non-confluence originates from unreachable states. The more powerful notion of *observable confluence* (Duck, Stuckey et al. 2007) takes reachability into account. Secondly, the standard notion of confluence is only applicable to terminating programs. Raiser and Tacchella (2007) extended the notion of confluence to non-terminating programs.

Haemmerlé and Fages (2007) develop a notion of abstract critical pairs for rewriting systems in general. They illustrate this notion for CHR's theoretical semantics. A particularly interesting result is that some traditional critical pairs can be disregarded because they are redundant.

Related Analyses. Abdennadher and Frühwirth (1998) showed how to do *completion* of CHR programs. Completion is a technique to transform a non-confluent program into a confluent one by adding rules. It allows extension, modification and specialization of existing programs.

A very useful notion is that of *operational equivalence* of two CHR programs. Two programs are operationally equivalent if for each query, the answer is the same (modulo variable renaming) according to each program. A straightforward extension of confluence, called *compatibility* of two programs, is shown to be too weak to capture the operational equivalence of CHR programs (Abdennadher and Frühwirth 1999; Abdennadher 2001). Instead, Abdennadher and Frühwirth (1999) give a decidable, sufficient, and necessary syntactic condition for operational equivalence of well-behaved (confluent and terminating) CHR *programs*. A sufficient syntactic condition is also given for the equivalence of two CHR *constraints*, defined in two different well-behaved CHR programs. The latter condition is also shown necessary for an interesting class of CHR programs.

Abdennadher and Frühwirth (2004) investigated the merging of two well-behaved CHR solvers. If the two programs are not compatible, well-behavedness can be regained by completion. Furthermore, Abdennadher and Frühwirth (2004) identify a class of solvers whose union is always confluent, and argue why finding a class whose union preserves termination is hard. Finally, Abdennadher and Frühwirth (2004) present a method to remove redundant rules from CHR programs, based on the notion of operational equivalence.

3.2 Termination

The first work on termination analysis of CHR programs was presented by Frühwirth (2000). Frühwirth demonstrated that termination proof techniques from logic programming and term rewrite systems can be adapted to the CHR context. Termination of CHR programs is proved by defining a ranking function from computation states to a well-founded domain such that the rank of consecutive computation states decreases. A condition on simplification rules guarantees such rank decreases for all consecutive states. This approach, however, cannot prove termination of CHR programs with propagation rules, because it is impossible to show decreases between consecutive states as these rules do not remove constraints from the store.

Recently, two new results on termination analysis of CHR were presented. Pilozzi, Schrijvers et al. (2007) describe a termination preserving transformation of CHR programs to Prolog programs. By reusing termination tools from logic programming and, indirectly, from term rewriting, proofs of termination of the transformed CHR programs are generated automatically, yielding the first fully automatic termination prover for CHR. The transformation, however, does not consider propagation histories. As such, it is applicable only to CHR programs without propagation rules. A transformation of single-headed propagation rules to equivalent simplification rules overcomes this problem partially.

The second contribution is presented by Voets, Pilozzi et al. (2007). Compare to previous approaches, theirsapproach is applicable to a much larger class of CHR programs. By formulating a new termination condition that verifies conditions imposed on the dynamic process of adding constraints to the store, they derive conditions for both simplification and propagation rules.

3.3 Complexity

For various CHR programs — general purpose programs as well as constraint solvers — an accurate, though rather ad hoc, complexity analysis has been made. We list the most notable examples in Section 3.3.1. While ad hoc methods give the most accurate results in practice, they cannot easily be generalized. Therefore, more structured approaches to complexity analysis have been proposed by means of meta-complexity theorems. An overview is given in Section 3.3.2.

3.3.1 Ad Hoc Analysis

A CHR implementation of the classical union-find algorithm was proven optimal by Schrijvers and Frühwirth (2006). Sneyers, Schrijvers et al. (2006a) showed the optimal complexity of an implementation of Dijkstra's shortest path algorithm that uses Fibonacci heaps. Frühwirth (2005a) formulated the complexity of a general-purpose lexicographical order constraint solver in terms of the number of ask and tell built-in constraints encountered during execution. Finally, Meister, Djelloul et al. (2006) derived the complexity of a solver for existentially quantified equations over finite and infinite trees, using bounds on the derivation length.

3.3.2 Meta-Complexity Results

Frühwirth (2001; 2002a; 2002b) investigated the time complexity of simplification rules for naive implementations of CHR. In this approach, a suitable termination order (called a *tight ranking*) is used as an upper bound on the derivation length. Combined with a worst-case estimate of the number and cost of rule application attempts, this results in a complexity meta-theorem which gives a rough upper bound of the time complexity. Recent work on optimizing compilation of CHR (cf. Section 4.2.2) allows meta-theorems that give much tighter complexity bounds. We now discuss two distinct approaches.

Ganzinger and McAllester (2002) propose a formalism called *Logical Algorithms* (LA) and prove a meta-complexity result. De Koninck, Schrijvers et al. (2007a) establish a close correspondence between CHR and LA (see also Section 6.1.3), allowing the LA meta-complexity result to be applied (indirectly) to a large class of CHR programs. De Koninck, Schrijvers et al. (2007a) actually address the meta-complexity of CHR[rp] programs, an extension of CHR discussed in Section 5.1.3. All CHR programs are also CHR[rp] programs. The Logical Algorithms approach was previously used, in a more ad hoc way, by Christiansen (2005) to derive the complexity of CHR grammars (see Section 7.3.3).

Sneyers, Schrijvers et al. (2009) explicitly decouple the two steps in the approach of Frühwirth (2002a; 2002b) by introducing abstract CHR machines. In the first step, the number of rule applications is estimated; this corresponds to the number of CHR machine steps. If a suitable termination order can be found, it can be used to show an upper bound. However for programs that are non-terminating in general, like a RAM machine simulator, or for which no suitable ranking can be found, other techniques have to be used to prove complexity properties. In the second

step, the complexity of rule application is computed for a given CHR program; this corresponds to simulating a CHR machine on a RAM machine. The first step depends only on the operational semantics of CHR, whereas the second step depends strongly on the performance of the code generated by the CHR compiler.

3.3.3 Complexity-wise Completeness of CHR

Sneyers, Schrijvers et al. (2009) also consider the space complexity of CHR programs. Some compiler optimizations like memory reuse (Sneyers, Schrijvers et al. 2006b) are crucial to achieve tight space complexity bounds (cf. Section 4.2.2). The most interesting result of Sneyers, Schrijvers et al. (2009) is the following "complexity-wise completeness" result for CHR, which implies that "everything can be done efficiently in CHR": For every algorithm (RAM machine program) which uses at least as much time as space, a CHR program exists which can be executed in the K.U.Leuven CHR system with time and space complexity within a constant from the original complexities. Complexity-wise completeness implies Turing completeness but is a much stronger property.

4 Systems and Implementation

CHR is first and foremost a programming language. Hence, a large part of CHR research has been devoted to the development of CHR systems and efficient execution of CHR programs. The two most comprehensive works on this subject are the Ph.D. theses of Duck (2005) and Schrijvers (2005). In this section, we provide an overview of their work as well as the many other contributions to the field.

4.1 Systems

Since the conception of CHR a large number of CHR systems (compilers, interpreters and ports) have been developed. In particular, in the last ten years the number of systems has exploded. Figure 5 presents a timeline of system development, branches and influences. We discuss these systems, grouped by host language or host paradigm, in more detail.

4.1.1 CHR(LP)

Logic Programming is the natural host language paradigm for CHR. Hence, it is not surprising that the CHR(Prolog) implementations are the most established ones. Holzbaur and Frühwirth (2000a) have laid the groundwork with their general compilation scheme for Prolog. This compilation scheme was first implemented in SICStus Prolog by Holzbaur, and later further refined in HAL by Holzbaur, García de la Banda et al. (2005) and in hProlog by Schrijvers and Demoen (2004b). The latter system, called the K.U.Leuven CHR system, was subsequently ported to many other Prolog systems and is currently available in XSB (Schrijvers, Warren et al. 2003; Schrijvers and Warren 2004), SWI-Prolog (Schrijvers, Wielemaker et al. 2005),
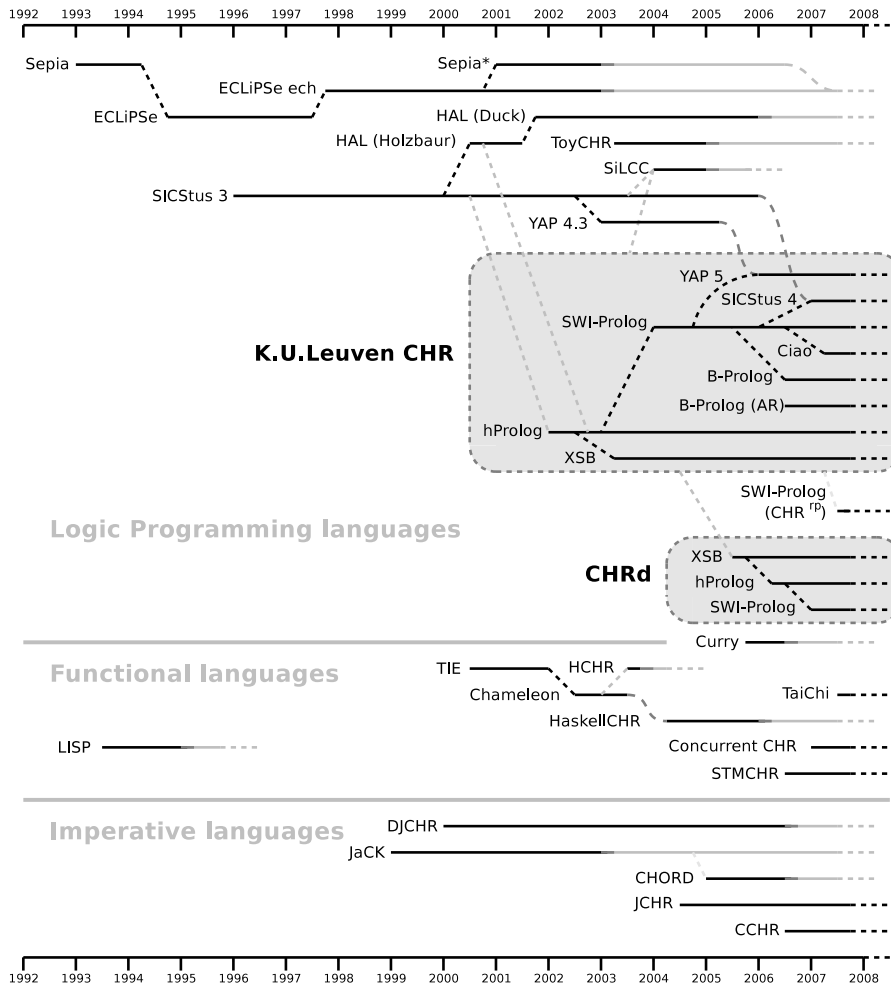
Fig. 5. A timeline of CHR implementations.

YAP, B-Prolog (using Action Rules; Schrijvers, Zhou et al. 2006), SICStus 4 and Ciao Prolog. Another system directly based on the work of Holzbaur and Schrijvers is the CHR library for SiLCC by Bouissou (2004). SiLCC is a programming language based on linear logic and concurrent constraint programming. All of these systems compile CHR programs to host language programs. The only available *interpreter* for CHR(Prolog) is TOYCHR[2].

Recently, systems with deviating operational semantics have been developed. The CHRd system by Sarna-Starosta and Ramakrishnan (2007) runs in XSB, SWI-Prolog and hProlog. It features a constraint store with set semantics and is particularly suitable for tabled execution. The CHR[rp] system by De Koninck, Stuckey et al. (2008) for SWI-Prolog provides rule priorities.

---

[2] by Gregory J. Duck, 2003. Download: `http://www.cs.mu.oz.au/~gjd/toychr/`

4.1.2 CHR(FP)

As type checking is one of the most successful applications of CHR in the context of Functional Programming (see Section 7.3.1), several CHR implementations were developed specifically for this purpose. Most notable is the Chameleon system (Stuckey and Sulzmann 2005) which features CHR as the programming language for its extensible type system. Internally, Chameleon uses the HaskellCHR implementation[3]. The earlier HCHR prototype (Chin, Sulzmann et al. 2003) had a rather heavy-weight and impractical approach to logical variables.

The aim of a 2007 Google Summer of Code project was to transfer this CHR based type checking approach to two Haskell compilers (YHC and nhc98). The project led to a new CHR interpreter for Haskell, called TaiChi (Boespflug 2007).

With the advent of software transactional memories (STM) in Haskell, two prototype systems with parallel execution strategies have been developed: STMCHR[4] and Concurrent CHR (Lam and Sulzmann 2007). These systems are currently the only known CHR implementations that exploit the inherent parallelism in CHR programs. Concurrent CHR also serves as the basis for Haskell-Join-Rules (Sulzmann and Lam 2007b) (cf. Section 6.1.2).

We also mention the Haskell library for the PAKCS implementation of the functional logic language Curry (Hanus 2006). The PAKCS system actually compiles Curry code to SICStus Prolog, and its CHR library is essentially a front-end for the SICStus Prolog CHR library. The notable added value of the Curry front-end is the (semi-)typing of the CHR code.

4.1.3 CHR(Java) and CHR(C)

Finally, CHR systems are available for both Java and C. These multiparadigmatic integrations of CHR and mainstream programming languages offer powerful synergetic advantages to the software developer: they facilitate the development of application-tailored constraint systems that cooperate efficiently with existing host language components. For a detailed discussion on the different conceptual and technical challenges encountered when embedding CHR into an imperative host language, we refer to Van Weert, Wuille et al. (2008).

CHR(Java). There are at least four implementations of CHR in Java. The earliest is the Java Constraint Kit (JaCK) by Abdennadher (2001) and others (Abdennadher, Krämer et al. 2002). It consists of three major components:

1. JCHR (Schmauß 1999) — a CHR dialect intended to resemble Java, in order to provide an intuitive programming experience. No operational semantics is specified for this system, and its behavior deviates from other CHR implementations.

---

[3] by Gregory J. Duck, 2004. Download: `http://www.cs.mu.oz.au/~gjd/haskellchr/`
[4] by Michael Stahl, 2007. Download: `http://www.cs.kuleuven.be/~dtai/projects/CHR/`

2. VisualCHR (Abdennadher and Saft 2001) — an interactive tool visualizing the execution of JCHR (cf. Section 4.3).
3. JASE (Krämer 2001) — a "Java Abstract Search Engine" in which tree-based search strategies can be specified. The JASE library is added to the JaCK framework as an orthogonal component. It provides a number of utility classes that aid the user to implement search algorithms in the Java host language. A typical algorithm consists of the following two operations, executed in a loop: a JCHR handler is run until it reaches a fix-point, after which a new choice is made. If an inconsistency is found, backtracking is used to return to the previous choice point. JASE aids in maintaining the search tree, and can be configured to use either trailing or copying.

DJCHR (Dynamic JCHR; Wolf 2001a) is an implementation of adaptive CHR (see Section 5.1.4). The incremental adaptation algorithm underlying DJCHR maintains *justifications* for rule applications and constraint additions. Wolf (2005) shows that these justifications, and in particular those of any derived `false` constraint, also serve as a basis for intelligent search strategies. As in JaCK, the different search algorithms are implemented orthogonally to the CHR program. Wolf's approach confirms that advanced search strategies are often more efficient than a low-level, built-in implementation of chronological backtracking (as in Prolog).

The K.U.Leuven JCHR system (Van Weert, Schrijvers et al. 2005) addresses the main issue of JaCK, its poor performance. The focus of K.U.Leuven JCHR is on both performance and integration with the host language. K.U.Leuven JCHR handlers integrate neatly with existing Java code, and it is currently one of the most efficient CHR systems available. The current implementation does not feature search capabilities.

Finally, the CHORD system (Constraint Handling Object-oriented Rules with Disjunctive bodies)[5], developed as part of the ORCAS project (Robin and Vitorino 2006), is a Java implementation of CHR$^\vee$ (Menezes, Vitorino et al. 2005).

CHR(C). CCHR (Wuille, Schrijvers et al. 2007) implements CHR for C. It is an extremely efficient CHR system conforming to the ω_r refined operational semantics. It uses a syntax that is intuitive to both CHR adepts and imperative programmers.

4.2 Compilation

Considerable research has been conducted on the efficient compilation of CHR. Section 4.2.1 provides an overview of the compilation schemes used by the different CHR systems; Sections 4.2.2 and 4.2.3 survey existing analyses and optimizations.

4.2.1 Compilation Schemes

The first CHR compilation scheme, for ECL^{i}PS^e Prolog, was described by Frühwirth and Brisset (1995). Holzbaur and Frühwirth (1999; 2000a) have adapted this scheme

---

[5] by Jairson Vitorino and Marcos Aurelio, 2005, `http://chord.sourceforge.net/`

from ECL^{i}PS^e's fairly specific suspension mechanism to the more primitive and flexible attributed variables feature found in SICStus Prolog. The latter form has been adopted by HALCHR, K.U.Leuven CHR, and was formalized in the refined operational semantics (Duck, Stuckey et al. 2004). A good overview of the compilation scheme can be found in the Ph.D. theses of Duck (2005) and Schrijvers (2005), and in (Van Weert, Wuille et al. 2008).

In its essence, the scheme maps each constraint to a procedure. Imposing the constraint then corresponds to calling the procedure. This procedure puts the new constraint in the constraint store datastructure, and attempts to fire rules involving the new constraint. For the latter purpose, the scheme contains an occurrence procedure for each occurrence of the constraint symbol in a rule. The main constraint procedure calls these occurrence procedures in the textual order of the rules. The reactivation of a constraint is realized through calling the occurrence procedures anew. Each procedure looks up the required additional constraints in the constraint store datastructure and checks both the guard and propagation history. If all tests succeed, the rule is committed to: an entry is added to the propagation history, the appropriate matching constraints are removed from the constraint store and the body of the rule is executed.

The Prolog compilation scheme is specifically designed for the built-in constraint theory of Herbrand equations. Duck, Stuckey et al. (2003) show how it can be extended to cover arbitrary constraint theories and solvers.

Schrijvers, Zhou et al. (2006) experimented with an action rules compilation scheme in BProlog. However, capturing the intricate reactivation behavior of CHR's refined operational semantics turned out to be hard because the action rules' behavior differs considerably on that account.

Lam and Sulzmann (2007) showed that software transactional memories (STM), as supported by the Glasgow Haskell Compiler, are a good match for the concurrent implementation of CHR. Sulzmann and Lam (2007a) also explored the use of Haskell's laziness and concurrency abstractions for implementing the search of partner constraints.

CHR(Java). Both JaCK (Schmauß 1999) and CHORD take a different approach compared to most other CHR compilers. Their front-end transforms the CHR source files to Java code that initializes the data structures of a generic runtime. CHR programs are then essentially interpreted. No major optimizations are performed.

DJCHR uses a compilation scheme similar to the basic CHR(Prolog) scheme, but extended with truth maintenance facilities required for adaptive constraint handling (Wolf 2001a). Justifications for constraints and rule applications are implemented efficiently using bit vectors. The runtime also implements adaptive unification and entailment algorithms. Following the approach of Holzbaur and Frühwirth (1999; 2000a), fast partner constraint retrieval is achieved using a form of attributed variables (Wolf 2001b).

K.U.Leuven JCHR and CCHR. The compilation schemes used by the K.U.Leuven JCHR and CCHR systems (Van Weert, Schrijvers et al. 2005; Wuille, Schrijvers

et al. 2007) are based on the basic compilation scheme for CHR(Prolog), modified to fit an imperative language (Van Weert, Wuille et al. 2008). Searching for partner constraints is done through explicit iteration. Also, the data structures required for the implementation of the constraint store are implemented more naturally and efficiently in an imperative host language than in an LP language. An important issue is that the host language typically cannot handle recursive calls efficiently. The common CHR(Prolog) scheme was therefore adjusted significantly to avoid frequent call stack overflows. The compilation scheme used by the K.U.Leuven JCHR system is described in detail in (Van Weert 2008).

The CCHR compiler performs some limited optimizations, like memory reuse, basic join ordering (see Section 4.2.2), as well as imperative-language specific optimizations. The K.U.Leuven JCHR compiler implements most of the optimizations mentioned in Section 4.2.2. For more details, see (Van Weert, Wuille et al. 2008).

CHR^{rp}. A compilation scheme for CHR$^{\text{rp}}$ that is strongly based on the one for regular CHR in Prolog, is presented in (De Koninck, Stuckey et al. 2008). It is formalized in the refined priority semantics ω_{rp}, which combines the refined operational semantics ω_r of regular CHR, with the priority semantics ω_p of CHR$^{\text{rp}}$.

The main differences are the following. The initial goal, as well as rule bodies, are executed in *batch mode*, i.e., no rule can fire as long as there are unprocessed goal or body constraints (i.e., the **Apply** transition is not allowed if the **Introduce** transition is applicable, cf. Fig. 4). New constraints are scheduled for activation at all priorities at which they have occurrences, instead of being activated as soon as they are processed. Constraints are activated at a given priority and as such only consider those rules that share this priority. Finally, after each rule firing, it is checked whether a scheduled constraint needs to be activated. To deal with so-called *dynamic* priority rules, which are rules for which the actual priority is only determined at runtime, a source-to-source transformation is given in (De Koninck, Stuckey et al. 2008), that transforms these rules to the desired form for the ω_{rp} semantics.

4.2.2 Analysis and Optimizing Compilation

A number of (mostly static) analyses and optimizations have been proposed to increase the performance of CHR systems (Holzbaur, García de la Banda et al. 2005; Schrijvers 2005; Duck 2005; Van Weert, Wuille et al. 2008). Without going into the technical details, we very briefly discuss a list of recent optimizations:

Indexing. The efficient, selective lookup of candidate partner constraints is indispensable for the efficient determination of matching rules. The traditional CHR(Prolog) compilation scheme (see Section 4.2.1) uses attributed variables for a constant time lookup of the constraints containing a known, unbound variable. Holzbaur, García de la Banda et al. (2005) propose the use of a balanced tree for the lookup of constraints via known ground arguments, which allows for logarithmic worst-case time lookup. Schrijvers (2005) further improves this using

19

hash tables to get amortized constant time constraint store operations. Sneyers, Schrijvers et al. (2006a) finally introduce array-based indexes to obtain correct space and time complexity guarantees (see also (Sneyers, Schrijvers et al. 2009)). Sarna-Starosta and Schrijvers (2007) show how indexing on compound term patterns is reduced to the above indexing techniques via program transformation.

Abstract interpretation. Schrijvers, Stuckey et al. (2005) present a general and systematic framework for program analysis of CHR for optimized compilation based on abstract interpretation. Two instances are given: late storage analysis (for reducing constraint store updates) and groundness analysis.

Functional dependencies. Functional dependency analysis (Duck and Schrijvers 2005) is a third instance of the abstract interpretation framework. It aims at tracking the cardinality of constraints (zero, one or more) for specializing constraint store indexes and related operations.

Guard optimization. The *guard optimization* (Sneyers, Schrijvers et al. 2005) removes redundant conjuncts in rule guards by reasoning on the ramifications of the refined operational semantics. More precisely, non-applicability of rules containing earlier removed occurrences of a constraint, can be used to infer redundant guard conditions.

Continuation optimization. The *continuation optimization* (Sneyers, Schrijvers et al. 2005) uses a similar reasoning to skip occurrences that can never lead to rule firings.

Delay avoidance. Schrijvers and Demoen (2004a) and Holzbaur, García de la Banda et al. (2005) describe techniques for avoiding the unnecessary delay and reactivation of constraints.

Memory reuse. Two memory optimizations (*in-place updates* and *suspension reuse*) were introduced by Sneyers, Schrijvers et al. (2006b). They significantly reduce the memory footprint and the time spent in garbage collection.

Join ordering. The time complexity of executing a CHR program is often determined by the join ordering — the order in which partner constraints are looked up in order to find matching rules. Holzbaur, García de la Banda et al. (2005) and Duck (2005) discussed ad-hoc heuristics for join ordering. De Koninck and Sneyers (2007) proposed a more rigorous investigation.

4.2.3 Code Specialization and Transformation

Most of the optimizations in the previous section are not expressible as source-to-source transformations. Like compiler optimization, program transformation can also be used to improve performance. The first proposal for CHR source-to-source transformation, by Frühwirth (2005c), adds redundant, specialized rules to a CHR program; these rules capture the effect of the original program for a particular goal. In more recent work, Tacchella, Gabbrielli et al. (2007) adapt the conventional notion of unfolding to CHR.

While the above study program transformation from a more theoretical point of view, Sarna-Starosta and Schrijvers (2007) show that various program transformation techniques improve indexing performance.

Frühwirth and Holzbaur (2003) propose to express CHR source-to-source transformations in CHR itself, and they show how to implement various language extensions (such as probabilistic CHR) by transformation to plain CHR. Van Weert, Sneyers et al. (2008) implemented an extension of CHR with aggregates (see Section 5.2.2) using a similar approach, with a more expressive transformation language.

4.3 Programming Environments

Over the past decade there has been an exponential increase in the number of CHR systems (Section 4.1), and CHR compilation techniques have matured considerably (Section 4.2). The support for advanced software development tools, such as debuggers, refactoring tools, and automated analysis tools, lags somewhat behind, and remains an important challenge for the CHR community.

VisualCHR (Abdennadher and Saft 2001), part of JaCK (see Section 4.1.3), is an interactive tool visualizing the execution of CHR rules. It can be used to debug and to improve the efficiency of constraint solvers.

Both Holzbaur's CHR implementation and the K.U.Leuven CHR system feature a trace-based debugger that is integrated in the Prolog four port tracer. A generic trace analysis tool, with an instantiation for CHR, is presented in (Ducassé 1999).

Checked type annotations are useful both for documentation and debugging purposes. CHR systems with a typed host language commonly perform type checking, or even type inference. The K.U.Leuven CHR system also allows optional type declarations, with both dynamic and static type checking. Coquery and Fages (2005) present a generic type system for CHR($\mathcal{H}$) (cf. also Section 7.3.1).

Ringwelski and Schlenker (2000a) propose the automatic inference of imported and exported symbols of CHR solvers, and the composition of solvers by matching up their interfaces (Ringwelski and Schlenker 2000b).

Schumann (2002) presents a literate programming system for CHR. The system allows for generating from the same literate program source both an algorithm specification typeset in LaTeX using mathematical notation, and the corresponding executable CHR source code.

Based on the theoretical results of Abdennadher, Frühwirth et al. (1999) (see Section 3.1), Bouissou (2004) implemented a confluence analyzer in CHR. Duck (2005) presents and evaluates a confluence checker based on the refined operational semantics. We do not know of any practical implementations of the other analyses of Section 3.1.

5 Extensions and Variants

Over the years, weaknesses and limitations of CHR have been identified, for instance regarding execution control, expressivity, modularity, incrementality, and search. In this section we consider extensions and variants of CHR that were proposed to tackle these issues.

5.1 Deviating Operational Semantics

We first discuss variants of CHR with an operational semantics that deviates from the commonly used refined operational semantics discussed in Section 2.2.2.

5.1.1 Probabilistic CHR

Probabilistic CHR (PCHR; Frühwirth, Di Pierro et al. 2002) extends CHR with probabilistic choice between the applicable rules in any state (though only for a given active constraint). It supports the implementation of algorithms like simulated annealing, which is often used for constrained optimization. It also gives rise to new concepts like probabilistic confluence and probabilistic termination. In PCHR, the probabilities are either a fixed number or an arithmetic expression involving variables that appear in the head. The latter are called *parametrised probabilities*. The actual probability that a rule is executed in a given state is found by dividing the rule probability by the sum of the probabilities of all fireable rule instances. Frühwirth, Di Pierro et al. (2002) give an implementation of PCHR by means of a source-to-source transformation using the framework proposed by Frühwirth and Holzbaur (2003).

As a simple example, the following PCHR program simulates tossing a coin:

```
toss(Coin) <=>0.5: Coin=head.
toss(Coin) <=>0.5: Coin=tail.
```

Given a `toss/1` constraint, one of the rules will be applied, with equal probability.

5.1.2 CHRd

The CHRd system (CHR with distributed constraint store) of Sarna-Starosta and Ramakrishnan (2007) alleviates the limitations of conventional CHR systems for efficient tabled evaluation encountered by Schrijvers and Warren (2004). For this purpose it implements a set-based operational semantics, i.e. the constraint store is a set rather than a multiset. Moreover, CHRd's constraint store has no global access point; constraints can only be retrieved through their logical variables. This rules out (the efficient execution of) of ground CHR programs.

5.1.3 CHR^{rp}

CHR^{rp} (De Koninck, Schrijvers et al. 2007b) is CHR extended with user-definable rule priorities. A rule's priority is either a number or an arithmetic expression involving variables that appear in the rule heads. The latter allows different instances of a rule to be executed at different priorities. The following CHR^{rp}-related topics are dealt with in other sections: its operational semantics in Section 2.2.3; its compilation schema in Section 4.2.1; and its implementation for SWI-Prolog in Section 4.1.1.

An example that illustrates the power of dynamic priorities is the following CHR^{rp} implementation of Dijkstra's algorithm:

```
1 :: source(V) ==> dist(V,0).
1 :: dist(V,D1) \ dist(V,D2) <=> D1 =< D2 | true.
D+2 :: dist(V,D), edge(V,C,W) ==> dist(W,D+C).
```

The priority of the last rule makes sure that new distance labels are propagated in the right order: first the nodes closest to the source node.

5.1.4 Adaptive CHR

Constraint solving in a continuously changing, dynamic environment often requires immediate adaptation of the solutions, i.e. when constraints are added or removed. By nature, CHR solvers already support efficient adaptation when constraints are added. Wolf (1999; 2000) introduces an extended incremental adaptation algorithm which is capable of adapting CHR derivations after constraint deletions as well. This algorithm is further improved by Wolf (2000a) with the elimination of local variables using *early projection*. An efficient implementation exists in Java (Wolf 2001a; 2001b; cf. Section 4.1.3).

Interesting applications of adaptive CHR include adaptive solving of soft constraints, discussed in Section 7.1, and the realization of intelligent search strategies, discussed in Sections 4.1.3 and 5.2.1.

5.2 Language Extensions

We now discuss some additional language features that have been added to the CHR language.

5.2.1 Disjunction and Search

Most constraint solvers require search next to constraint simplification and propagation. However pure CHR does not offer any support for search. Abdennadher and Schütz (1998) propose a solution to this problem: an extension of CHR with disjunctions in rule bodies (see also Abdennadher 2000; 2001). The resulting language is denoted CHR$^\vee$ (pronounced "CHR-or"), and is capable of expressing several declarative evaluation strategies, including bottom-up evaluation, top-down evaluation, model generation and abduction (see Section 7.3.2 for abduction). Any (pure) Prolog program can be rephrased as an equivalent CHR$^\vee$ program (Abdennadher 2000; 2001). An interesting aspect of CHR$^\vee$ is that the extension comes for free in CHR(Prolog) implementations by means of the built-in Prolog disjunction and search mechanism.

As a typical example of programming in CHR$^\vee$, consider the following rule:

```
labeling, X::Domain <=> member(X,Domain), labeling.
```

Note the implicit disjunction in the call to the Prolog predicate member/2.

Various ways have been proposed to make the search in CHR$^\vee$ programs more flexible and efficient. Menezes, Vitorino et al. (2005) present a CHR$^\vee$ implementation for Java in which the search tree is made explicit and manipulated at runtime

to improve efficiency. The nodes in the search tree can be reordered to avoid redundant work. De Koninck, Schrijvers et al. (2006b) extend both the theoretical and refined operational semantics of CHR towards CHR$^\vee$. The theoretical version leaves the search strategy undetermined, whereas the refined version allows the specification of various search strategies. In the same work, an implementation for different strategies in CHR(Prolog) is realized by means of a source-to-source transformation.

For CHR(Java) systems, unlike for CHR(LP) systems, the host language does not provide search capabilities. The flexible specification of intelligent search strategies has therefore received considerable attention in several CHR(Java) systems (Krämer 2001; Wolf 2005). As described in Section 4.1.3, in these systems, the search strategies are implemented and specified in the host language itself, orthogonally to the actual CHR program. Wolf, Robin et al. (2007) propose an implementation of CHR$^\vee$ using the ideas of (Wolf 2005), in order to allow a more declarative formulation of search in the bodies of CHR rules, while preserving efficiency and flexibility. A refined operational semantics of the proposed execution strategy is presented as well. Along these lines is the approach proposed by Robin, Vitorino et al. (2007), where disjunctions in CHR$^\vee$ are transformed into special purpose constraints that can be handled by an external search component such as JASE (Krämer 2001).

5.2.2 Negation and Aggregates

CHR programmers often want to test for the absence of constraints. CHR was therefore extended with *negation as absence* by Van Weert, Sneyers et al. (2006). Negation as absence was later generalized to a much more powerful language feature, called *aggregates* (Sneyers, Van Weert et al. 2007). Aggregates accumulate information over unbounded portions of the constraint store. Predefined aggregates include `sum`, `count`, `findall`, and `min`. The proposed extension also features nested aggregate expressions over guarded conjunctions of constraints, and application-tailored user-defined aggregates. Aggregates lead to increased expressivity and more concise programs. An implementation based on source-to-source transformations (Van Weert, Sneyers et al. 2008) is available. The implementation uses efficient incremental aggregate computation, and empirical results show that the desired runtime complexity is attainable with an acceptable constant time overhead.

As an example of nested aggregate expressions, consider the following rule:

```
eulerian, forall(node(N),(
             count(edge(N,_),X), count(edge(_,N),X)
         )) <=> writeln(graph_is_eulerian).
```

The above rule is applicable if for every constraint `node(N)`, the number of outgoing edges in `N` equals the number of incoming edges (i.e. if the first number is `X`, the other number must also be `X`).

5.3 Solver Hierarchies

While the theory of the CHR language generally considers arbitrary built-in solvers, traditional CHR implementations restrict themselves to the Herbrand equality constraint solver, with very little, if any, support for other constraint solvers.

Duck, Stuckey et al. (2003) show how to build CHR solvers on top of arbitrary built-in constraint solvers by means of *ask* constraints. The ask constraints signal the CHR solver when something has changed in the built-in store with respect to variables of interest. Then the relevant CHR constraints may be reactivated.

Schrijvers, Demoen et al. (2006) provide an automated means for deriving ask versions of CHR constraints. In this way full hierarchies of constraint solvers can be written in CHR, where one CHR solver serves as the built-in solver for another CHR solver.

6 Relation to Other Formalisms

The relation of CHR to other formalisms has recently received quite a lot of attention. In this section we give a brief overview.

As a general remark, we should mention that most of the formalisms related to CHR are limited to ground terms and lack the equivalent of propagation rules. In that sense, they are subsumed by CHR. Also, CHR can be seen as an instance or a generalization of concurrent constraint programming, constraint logic programming, constraint databases, and deductive databases.

Logical formalisms. In Section 2.1, we discussed the logical semantics of CHR. CHR can be given a classical logic semantics (Frühwirth 1998), a linear logic semantics (Betz and Frühwirth 2005; 2007), and a transaction logic semantics (Meister, Djelloul et al. 2007). Also, it should be noted that CHR$^\vee$ subsumes Prolog. Frame-Logic (F-Logic) is an object-oriented extension of classical predicate logic. Käser and Meister (2006) explore the relation between CHR and F-Logic by implementing (a fragment of) F-Logic in CHR.

Term rewriting. CHR can be considered as associative and commutative (AC) term rewriting of flat conjunctions. The term rewriting literature inspired many results for CHR, for example on confluence (see Section 3.1) and termination (see Section 3.2). Recently, Duck, Stuckey et al. (2006) proposed the formalism of ACD term rewriting, which subsumes both AC term rewriting and CHR.

6.1 Set-Based Formalisms

Numerous formalisms have been proposed that are based on (multi-)set rewriting.

6.1.1 Production Rules / Business Rules

Production rules, or business rules as they are now often called, is a rule-based paradigm closely related to CHR. Classic matching algorithms, such as RETE and

25

LEAPS, have influenced early work on CHR compilation. Production rules have also inspired the research towards extending CHR with aggregates (see Section 5.2.2).

6.1.2 Join-Calculus

The join-calculus is a calculus for concurrent programming, with both stand-alone implementations and extensions of general purpose languages, such as JoCaml (OCaml), Join Java and Polyphonic C#.

Sulzmann and Lam (2007b) propose a Haskell language extension for supporting join-calculus-style concurrent programming, based on CHR. Join-calculus rules, called chords, are essentially guardless simplification rules with linear match patterns. In a linear pattern, different head conjuncts are not allowed to share variables. Hence, CHR offers considerably increased expressivity over the join-calculus: propagation rules, general guards and non-linear patterns.

6.1.3 Logical Algorithms

As already mentioned in Section 3.3.2, CHR is strongly related to the Logical Algorithms (LA) formalism by Ganzinger and McAllester (2002). De Koninck, Schrijvers et al. (2007a) have showed how LA programs can easily be translated into CHR^{rp} programs. The opposite only holds for a subset of CHR^{rp} since the LA language lacks the ability to plug in any built-in constraint theory, and also only supports ground constraints (called *assertions* in LA terminology). The correspondence between both languages makes it possible to apply the meta-complexity result for LA to a subset of CHR^{rp} as explained in Section 3.3.2. It is also interesting that the first actual implementation of LA is that of De Koninck, Schrijvers et al. (2007a), which compiles the language into (regular) CHR rules.

6.1.4 Equivalent Transformation Rules

The Equivalent Transformation (ET) computation model is a rewriting system which consists of the application of conditional multi-headed multi-body ET rules (ETR). Although CHR and ETR are similar in syntax, they have different theoretical bases: CHR is based on logical equivalence of logical formulas, whereas ET is based on the set equivalence of descriptions. Shigeta, Akama et al. (2006) investigate the relation between CHR and ETR.

6.2 Graph-Based Formalisms

Recently, CHR has also been related to a number of graph-based formalisms:

6.2.1 Graph Transformation Systems

Raiser (2007) describes an elegant embedding of Graph Transformation Systems (GTS) in CHR. The confluence properties (see Section 3.1) for CHR and GTS are

similar; in particular, a sufficient criterion for confluence of a GTS is the confluence of the corresponding CHR program. Using a slightly weaker notion of confluence of CHR (in the spirit of observable confluence; Duck, Stuckey et al. 2007), standard CHR confluence checkers can be reused to decide GTS confluence.

6.2.2 Petri Nets

Petri nets are a well-known formalism for the modeling and analysis of concurrent processes. Betz (2007) provides a first study of the relation between CHR and Petri nets. He provides a sound and complete translation of place/transition nets (P/T nets) — a standard variant of Petri nets — into a small segment of CHR. P/T nets are, unlike CHR (cf. Section 3.3), not Turing complete. A translation of a significant subsegment of CHR into *colored* Petri nets is presented as well by Betz (2007). This work is a promising first step towards cross-fertilization between both formalisms. Results from Petri nets could for instance be applied to analyze concurrency properties of CHR programs.

6.2.3 LMNtal

LMNtal (Ueda et al. 2006) is a language based on hierarchical graph rewriting which intends to unify constraint-based concurrency and CHR. It uses logical variables to represent connectivity and so-called *membranes* to represent hierarchy. Flat LMNtal rules (rules without membranes) can be seen as simplification rules in CHR.

7 Applications

The main application domain considered in the previous CHR survey (Frühwirth 1998) is the development of constraint solvers. It also discusses two other applications of CHR in some depth. The first one is related to the problem of finding an optimal placement of wireless transmitters (Frühwirth and Brisset 1998; 2000); the second one is an expert system for estimating the maximum fair rent in the city of Munich, called the *Munich Rent Advisor* (Frühwirth and Abdennadher 2001). In this section, we give an overview of more recent applications of CHR.

7.1 Constraint Solvers

CHR was originally designed specifically for writing constraint solvers. We discuss some recent examples of constraint solvers written in CHR. The following examples illustrate how CHR can be used to build effective prototypes of non-trivial constraint solvers:

Lexicographic order. Frühwirth (2006a) presented a constraint solver for a lexicographic order constraint in terms of inequality constraints offered by the underlying solver. The approach is general in that it can be used for any constraint domain offering inequality (*less-than*) constraints between problem variables. Still, the program is very concise and elegant; it consists of just six rules.

Rational trees. Meister, Djelloul et al. (2006) presented a solver for existentially quantified conjunctions of non-flat equations over rational trees. The solver consists of a transformation to flat equations, after which a classic CHR solver for rational trees can be used. This results in a complexity improvement with respect to previous work. In (Djelloul, Dao et al. 2007), the solver for rational trees is used as part of a more general solver for (quantified) first-order constraints over finite and infinite trees.

Sequences. Kosmatov (2006a; 2006b) has constructed a constraint solver for sequences, inspired by an earlier solver for lists by Frühwirth. The solver expresses many sequence constraints in terms of two basic constraints, for sequence concatenation and size.

Non-linear constraints. A general purpose CHR-based CLP system for non-linear (polynomial) constraints over the real numbers was presented by De Koninck, Schrijvers et al. (2006a). The system, called $INCLP(\mathbb{R})$, is based on interval arithmetic and uses an interval Newton method as well as constraint inversion to achieve respectively box and hull consistency.

Interactive constraint satisfaction. Alberti, Gavanelli et al. (2005) describe the implementation of a CLP language for expressing Interactive Constraint Satisfaction Problems (ICSP). In the ICSP model incremental constraint propagation is possible even when variable domains are not fully known, performing acquisition of domain elements only when necessary.

Solvers derived from union-find. Frühwirth (2006b) proposes linear-time algorithms for solving certain boolean equations and linear polynomial equations in two variables. These solvers are derived from the classic union-find algorithm (see Section 7.2).

In the rest of this subsection we discuss, in a bit more detail, some typical application domains in which CHR has been used to implement constraint solvers.

7.1.1 Soft Constraints and Scheduling

An important class of constraints are the so-called *soft constraints* which are used to represent preferences amongst solutions to a problem. Unlike hard (required) constraints which must hold in any solution, soft (preferential) constraints must only be satisfied as far as possible. Bistarelli, Frühwirth et al. (2004) present a series of constraint solvers for (mostly) extensionally defined finite domain soft constraints, based on the framework of *c-semirings*. In this framework, soft constraints can be combined and projected onto a subset of their variables, by using the two operators of a c-semiring. A node and arc consistency solver is presented, as well as complete solvers based on variable elimination or branch and bound optimization.

Another well-known formalism for describing over-constrained systems is that of *constraint hierarchies*, where constraints with hierarchical strengths or preferences can be specified, and non-trivial error functions can be used to determine the solutions. Wolf (2000b) proposes an approach for solving dynamically changing constraint hierarchies. Constraint hierarchies over finite domains are transformed

into equivalent constraint systems, which are then solved using an adaptive CHR solver (Wolf, Gruenhagen et al. 2000; Wolf 2001a; see Section 5.1.4).

Scheduling. Abdennadher and Marte (2000) have successfully used CHR for scheduling courses at the university of Munich. Their approach is based on a form of soft constraints, implemented in CHR, to deal with teacher's preferences. A related problem, namely that of assigning classrooms to courses given a timetable, is dealt with by Abdennadher, Saft et al. (2000). An overview of both applications is found in (Abdennadher 2001).

7.1.2 Spatio-Temporal Reasoning

In the context of autonomous mobile robot navigation, a crucial research topic is automated qualitative reasoning about spatio-temporal information, including orientation, named or compared distances, cardinal directions, topology and time. The use of CHR for spatio-temporal reasoning has received considerable research attention. We mention in particular the contributions of Escrig et al. (Escrig and Toledo 1998a; 1998b; Cabedo and Escrig 2003).

Meyer (2000) has applied CHR for the constraint-based specification and implementation of diagrammatic environments. Grammar-based specifications of diagrammatic objects are translated to directly executable CHR rules. This approach is very powerful and flexible. The use of CHR allows the integration with other constraint domains, and additional CHR rules can easily be added to model more complex diagrammatic systems. Similar results are obtained with CHRG in the context of natural language processing (see Section 7.3.3).

7.1.3 Multi-Agent Systems

FLUX (Thielscher 2002; 2005) is a high-level programming system, implemented in CHR and based on fluent calculus, for cognitive agents that reason logically about actions in the context of incomplete information. An interesting application of this system is FLUXPLAYER (Schiffel and Thielscher 2007), which won the 2006 General Game Playing (GGP) competition at AAAI'06. Seitz, Bauer et al. (2002) and Alberti et al. (2004; 2004; 2006) also applied CHR in the context of multi-agent systems.

Lam and Sulzmann (2006) explore the use of CHR as an agent specification language, founded on CHR's linear logic semantics (see Section 2.1.2). They introduce a monadic operational semantics for CHR, where special *action constraints* have to be processed in sequence. They reason about the termination and confluence properties of the resulting language. They were also the first to propose the use of invariants for CHR confluence testing, an approach that was later formalized by Duck, Stuckey et al. (2007) — cf. Section 3.1.

7.1.4 Semantic Web and Web 3.0

One of the core problems related to the so-called *Semantic Web* is the integration and combination of data from diverse information sources. Bressan and Goh (1998) describe an implementation of the COIN (COntext INterchange) mediator that uses CHR for solving integrity constraints. In more recent work, CHR is used for implementing an extension of the COIN framework, capable of handling more data source heterogeneity (Firat 2003). Badea, Tilivea et al. (2004) present an improved mediator-based integration system. It allows forward propagation rules involving model predicates, whereas COIN only allows integrity constraints on source predicates.

The Web Ontology Language (OWL) is based on Description Logic (DL). Various rule-based formalisms have been considered for combination and integration with OWL or other description logics. Frühwirth (2007) proposes a CHR-based approach to DL and DL rules. Simply encoding the first-order logic theory of the DL in CHR results in a concise, correct, confluent and concurrent CHR program with performance guarantees.

The Cuypers Multimedia Transformation Engine (Geurts, van Ossenbruggen et al. 2001) is a prototype system for automatic generation of Web-based presentations adapted to device-specific capabilities and user preferences. It uses CHR and traditional CLP to solve qualitative and quantitative spatio-temporal constraints.

7.1.5 Automatic Generation of Solvers

Many authors have investigated the automatic generation of CHR rules, constituting a constraint solver, from a formal specification. Most authors consider extensionally defined constraints over (small) finite domains as the specification.

A first line of work is that of Apt and Monfroy (2001). From an extensional definition of a finite domain constraint, a set of propagation rules is derived. These rules reduce the domain of one variable based on the domains of the other variables involved in the same constraint. As an extension, Brand and Monfroy (2003) propose to transform the derived rules to obtain stronger propagation rules. This technique is useful to obtain derived versions of constraints, such as the conjunction of two constraints.

A second line of work is that of Abdennadher and Rigotti. In (2004) they derive propagation rules from extensionally defined constraints. There are two main differences with the previous line of work. Firstly, the rules are assembled from given parts, and, secondly, propagation rules are transformed into simplification rules when valid. This algorithm is implemented by the Automatic Rule Miner tool (Abdennadher, Olama et al. 2006). They extend their approach to intensional constraint definitions, where constraints are defined by logic programs (2005), and further to symbolically derive rules from the logic programs, rather than from given parts (2008).

Brand (2002) has proposed a method to eliminate redundant propagation rules

and applies it to rules generated by RuleMiner: one of the algorithms that forms the basis of the Automatic Rule Miner tool by Abdennadher et al. (2006).

7.2 Union-Find and Other Classic Algorithms

CHR is used increasingly as a general-purpose programming language. Starting a trend of investigating this side of CHR, Schrijvers and Frühwirth (2005a; 2006) implemented and analyzed the classic union-find algorithm in CHR. In particular, they showed how the optimal complexity of this algorithm can be achieved in CHR — a non-trivial achievement since this is believed to be impossible in pure Prolog. This work lead to parallel versions of the union-find algorithm (Frühwirth 2005b) and several derived algorithms (Frühwirth 2006b). Inspired by the specific optimal complexity result for the union-find algorithm, Sneyers, Schrijvers et al. (2009) have generalized this to arbitrary (RAM-machine) algorithms (see Section 3.3.2).

The question of finding elegant and natural implementations of classic algorithms in CHR remains nevertheless an interesting research topic. Examples of recent work in this area are implementations of Dijkstra's shortest path algorithm using Fibonacci heaps (Sneyers, Schrijvers et al. 2006a) and the preflow-push maximal flow algorithm (Meister 2006).

7.3 Programming Language Development

Another application area in which CHR has proved to be useful is the development of programming languages. CHR has been applied to implement additional programming language features such as type systems (Section 7.3.1), meta-programming (Section 7.3.4), and abduction (Section 7.3.2). Also, CHR has been used to implement new programming languages, especially in the context of computational linguistics (Section 7.3.3). Finally, CHR has been used for testing and verification (Section 7.3.5).

7.3.1 Type Systems

CHR's aptness for symbolic constraint solving has led to many applications in the context of type system design, type checking and type inference. While the basic Hindley-Milner type system requires no more than a simple Herbrand equality constraint, more advanced type systems require custom constraint solvers.

Alves and Florido (2002) presented the first work on using Prolog and CHR for implementing the type inference framework HM(X), i.e. type inference for extensions of the Hindley-Milner type system. This work was followed up by TypeTool (Simões and Florido 2004), a tool for visualizing the type inference process.

The most successful use of CHR in this area is for Haskell type classes. Type classes are a principled approach to ad hoc function overloading based on type-level constraints. By defining these type class constraints in terms of a CHR program (Stuckey and Sulzmann 2005) the essential properties of the type checker (soundness, completeness and termination) can easily be established. Moreover, various

extensions, such as multi-parameter type classes (Sulzmann, Schrijvers et al. 2006) and functional dependencies (Sulzmann, Duck et al. 2007) are easily expressed. At several occasions Sulzmann argues for HM(CHR), where the programmer can directly implement custom type system extensions in CHR. Wazny (2006) has made considerable contributions in this setting, in particular to type error diagnosis based on justifications.

Coquery and Fages (2003; 2005) presented TCLP, a CHR-based type checker for Prolog and CHR(Prolog) that deals with parametric polymorphism, subtyping and overloading. Schrijvers and Bruynooghe (2006) reconstruct type definitions for untyped functional and logic programs.

Finally, Chin, Craciun et al. (2006) presented a control-flow-based approach for variant parametric polymorphism in Java.

7.3.2 Abduction

Abduction is the inference of a cause to explain a consequence: given B determine A such that $A \rightarrow B$. It has applications in many areas: diagnosis, recognition, natural language processing, type inference, ...

The earliest paper connecting CHR with abduction is that of Abdennadher and Christiansen (2000). It shows how to model logic programs with abducibles and integrity constraints in $CHR^\vee$. The disjunction is used for (naively) enumerating the alternatives of the abducibles, while integrity constraints are implemented as propagation rules. The HYPROLOG system of Christiansen and Dahl (2005a) combines the above approach to abductive reasoning with abductive-based logic programming in one system. Both the abducibles and the assumptions are implemented as CHR constraints. Christiansen (2006) also proposes the use of CHR for the implementation of *global abduction*, an extended form of logical abduction for reasoning about a dynamic world.

Gavanelli, Lamma et al. (2003) propose two variant approaches to implementing abductive logic programming. The first is similar to the above approach, but tries to leverage as much as possible from a more efficient boolean constraint solver, rather than CHR. The second approach propagates abducibles based on integrity constraints.

The system of Alberti, Chesani et al. (2005) extends the abductive reasoning procedure with the dynamic acquisition of new facts. These new facts serve to confirm or disconfirm earlier hypotheses.

Sulzmann, Wazny et al. (2005) show that advanced type system features give rise to implications of constraints, or, in other words, constraint abduction. An extension of their CHR-based type checking algorithm is required to deal with these implications.

7.3.3 Computational Linguistics

CHR allows flexible combinations of top-down and bottom-up computation (Abdennadher and Schütz 1998), and abduction fits naturally in CHR as well (see

Section 7.3.2). It is therefore not surprising that CHR has proven a powerful implementation and specification tool for language processors.

Penn (2000) focuses on another benefit CHR provides to computational linguists, namely the possibility of delaying constraints until their arguments are sufficiently instantiated. As a comprehensive case study he considers a grammar development system for HPSG, a popular constraint-based linguistic theory.

Morawietz and Blache (2002) show that CHR allows a flexible and perspicuous implementation of a series of standard chart parsing algorithms (cf. also Morawietz (2000)), as well as more advanced grammar formalisms such as minimalist grammars and property grammars. Items of a conventional chart parser are modeled as CHR constraints, and new constraints are deduced using constraint propagation. The constraint store represents the chart, from which the parse tree can be determined. Along the same lines is the CHR implementation of a context-sensitive, rule-based grammar formalism by Garat and Wonsever (2002).

A more recent application of CHR in the context of natural language processing is (Christiansen and Have 2007), where a combination of Definite Clause Grammars (DCG) and CHR is used to automatically derive UML class diagrams from use cases written in a restricted natural language.

CHR Grammars. The most successful approach to CHR-based language processing is given by CHR grammars (CHRG), a highly expressive, bottom-up grammar specification language proposed by Christiansen (2005). Contrary to the aforementioned approaches, which mostly use CHR as a general-purpose implementation language, Christiansen recognizes that the CHR language itself can be used as a powerful grammar formalism. CHRG's, built as a relatively transparent layer of syntactic sugar over CHR, are to CHR what DCG's are to Prolog.

CHRG's inherent support for context-sensitive rules readily allows linguistic phenomena such as long-distance reference and coordination to be modeled naturally (Christiansen 2005; Aguilar-Solis and Dahl 2004; Dahl 2004). CHRG grammar rules can also use extra-grammatical hypotheses, modeled as regular CHR constraints. This caters, e.g., for straightforward implementations of assumption grammars and abductive language interpretation with integrity constraints.

Applications of CHRG. Using CHRG, Dahl and Blache (2005) develop directly executable specifications of property grammars. They show this combination of grammar formalisms to be robust, and able to handle various levels of granularity, as well as incomplete and incorrect input. In (Dahl and Gu 2006), an extension of this approach is used to extract concepts and relations from biomedical texts.

Dahl and Voll (2004) generalize the property grammar parsing methodology into a general concept formation system, providing a cognitive sciences view of problem solving. Applications of this formalism include early lung cancer diagnosis (Barranco-Mendoza 2005, Chapter 4), error detection and correction of radiology reports obtained from speech recognition (Voll 2006, Section 5.2.8), and the analysis of biological sequences (Bavarian and Dahl 2006).

Christiansen and Dahl (2003) use an abductive model based on CHRG to di-

agnose and correct grammatical errors. Other applications of CHRG include the characterization of the grammar of ancient Egyptian hieroglyphs (Hecksher, Nielsen et al. 2002), linguistic discourse analysis (Christiansen and Dahl 2005b), and the disambiguation of biological text (Dahl and Gu 2007). An approach similar to CHRG is taken by Bès and Dahl (2003) for the parsing of balanced parentheses in natural language.

7.3.4 Meta-Programming

Christiansen and Martinenghi (2000) develops a meta-programming environment, DemoII, that relies on CHR for its powerful features. Firstly, the meta-interpreter is made reversible in order to both evaluate queries and generate programs. Secondly, soundness of negation-as-failure is achieved through incremental evaluation.

7.3.5 Testing and Verification

Another application domain for which CHR has proved useful is software testing and verification. Ribeiro, Zúquete et al. (2000) present a CHR-based tool for detecting security policy inconsistencies. Lötzbeyer and Pretschner (2000) and Pretschner, Slotosch et al. (2004) propose a model-based testing methodology, in which test cases are automatically generated from abstract models using CLP and CHR. They consider the ability to formulate arbitrary test case specifications by means of CHR to be one of the strengths of their approach. Gouraud and Gotlieb (2006) use a similar approach for the automatic generation of test cases for the Java Card Virtual Machine (JCVM). A formal model of the JCVM is automatically translated into CHR, and the generated CHR program is used to generate test cases.

More of an exploration than testing application is the JmmSolve framework (Schrijvers 2004). Its purpose is to explore and test the behavior of declarative memory models for Java, based on the Concurrent Constraint-based Memory Machines proposal of V. Saraswat.

7.4 Industrial CHR Users

Although most CHR systems are essentially still research prototypes, there are a few systems that can be considered to be robust enough for industrial application. We give a few examples of companies that are currently using CHR.

The New-Zealand-based company Scientific Software & Systems Ltd. (2008) is one of the main industrial users of CHR. The company uses CHR throughout its flagship product the SecuritEase stock broking system[6]. SecuritEase provides front office (order entry) and back-office (settlement and delivery) functions for stock brokers in Australia and New Zealand. Inside SecuritEase CHR is used for:

1. implementing the logic to recognize advantageous market conditions to automatically place orders in equity markets,

---

[6] http://www.securitease.com/

2. translating high-level queries to SQL,

3. describing complex relationships between mutually dependent fields on user input screens, and calculating the consequences of user input actions, and

4. realizing a Financial Information eXchange (FIX) server.

The Canadian company Cornerstone Technology Inc.[7] has created an inference engine for solving and optimizing collections of design constraints, using Prolog and CHR. The design constraints work together to determine what design configuration to use, select components from catalogs, compute dimensions for custom components, and arrange the components into assemblies. The engine allows for generating, interactive editing, and validating of injection mould designs. Part of the system is covered by US Patent 7,117,055.

BSSE System and Software Engineering[8], a German company specializing in the discipline of full automation of software development, uses K.U.Leuven JCHR for the generation of test data for unit tests.

At the MITRE Corporation[9], CHR is used in the context of optical network design. It is used to implement constraint-based optimization, network configuration analysis, and as a tool coordination framework.

8 Conclusions

In this section, we first try to assess to what extent we have covered the CHR literature in this survey (Section 8.1). Next, in Section 8.2, we look back at the research topics that were mentioned in the previous CHR survey (Frühwirth 1998) as being open issues. Finally, to conclude this survey, we propose four remaining "grand challenges" for CHR researchers.

8.1 Survey Coverage and Bibliographic Meta-Information

This survey cites 182 publications related to CHR. For convenience, we define the total number of CHR-related publications as the number of publications that cite (Frühwirth 1998), according to Google Scholar (with some manual corrections for errors in the Google Scholar result list). Figure 6 compares the number of publications we cite in this survey with the total number of CHR-related publications since 1998. Globally, we cite roughly half of the CHR-related publications; the remaining half are preliminary versions of cited publications and papers that use CHR or refer to CHR in only a relatively minor way.

Figure 7 shows how CHR authors have collaborated. This graph is derived from the joint authorships of papers cited in this survey.

---

[7] `http://www.cornerstonemold.com/`
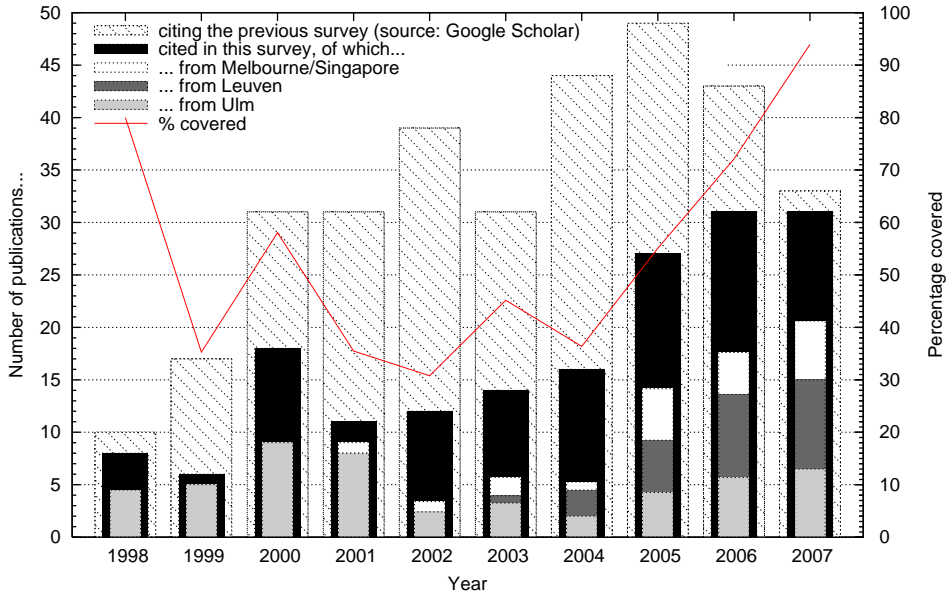[8] `http://www.bsse.biz/`
[9] `http://www.mitre.org/`

Fig. 6. Estimation of the number of CHR-related publications and the number of CHR-related publications cited in this survey (and their origin).

8.2 Retrospection

The first CHR survey (Frühwirth 1998) ended with a list of research topics from the first draft paper on CHR in 1991, noting that most of those topics were still open in 1998. We re-examine that list:

- **Termination and confluence.** As seen in Sections 3.1 and 3.2, both confluence (Abdennadher, Frühwirth et al. 1999; Duck 2005; Duck, Stuckey et al. 2007; Raiser and Tacchella 2007) and termination (Frühwirth 2000; Pilozzi, Schrijvers et al. 2007; Voets, Pilozzi et al. 2007) have received a great deal of attention in the past ten years. While substantial results have been obtained for confluence, this notion turned out to be rather impractical. The recent proposal of observable confluence seems to be a first step towards a more practical notion of confluence. In the area of termination analysis there appears to be a much wider scope for improvement. Existing work, mostly carrying over results of other programming languages, only works for a limited fraction of programs. Propagation rules and logical variables, part of typical CHR programs, cannot be dealt with. An important breakthrough is still ahead of us.
- **Negation and entailment of constraints.** Negation as absence (Van Weert, Sneyers et al. 2006) was recently explored (see Section 5.2.2), but this has little relation to the logical negation of constraints. The topic of entailment is closely related to building solver hierarchies (Duck, Stuckey et al. 2003; Schrijvers, Demoen et al. 2006) (see Section 5.3). Both topics are still an important issue today.
- **Combination and communication of solvers.** This topic is again related to solver hierarchies (see the item above), but also to integration of solvers (Abden-
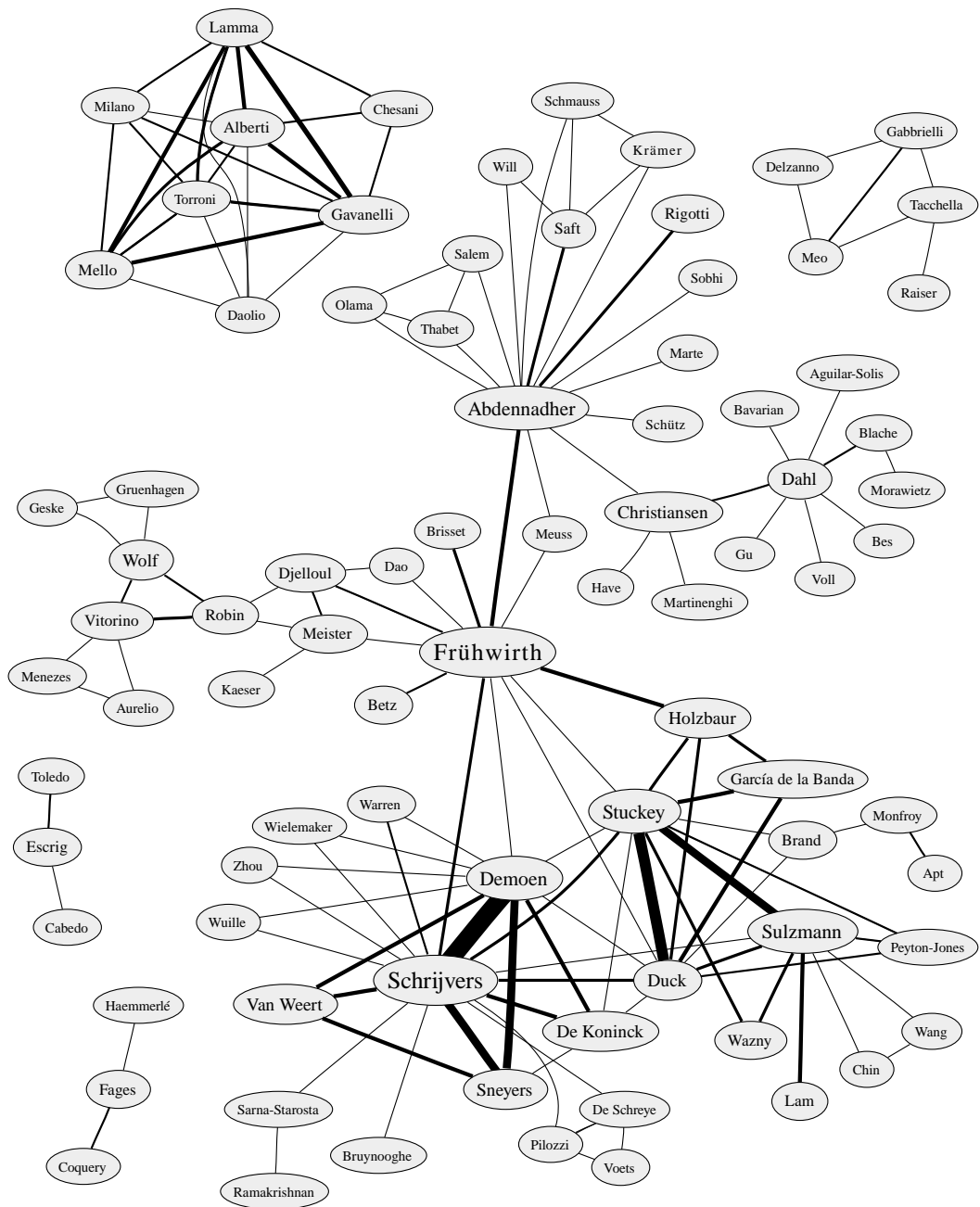
36

Fig. 7. CHR author collaboration graph. Two authors are connected if they have co-authored a CHR-related paper cited in this survey. The number of co-authored papers is reflected in the edge thickness.

nadher and Frühwirth 2004) (see Section 3.1). Another potential approach could be based on a compositional semantics for CHR (Delzanno, Gabbrielli et al. 2005) (see Section 2.2). In any case, this is still an important open research topic.

- **Correctness w.r.t. specifications, debugging.** Although some progress has been made (Section 4.3), these topics are still mostly open.
- **Soft constraints with priorities.** As we discussed in Section 7.1, two distinct approaches were proposed for dealing with (prioritized) soft constraints in CHR.
- **Dynamic constraints, removable constraints.** Wolf, Gruenhagen et al. (2000) designed an incremental adaptation algorithm that supports dynamic and removable constraints. An efficient implementation of adaptive CHR exists for Java (Wolf 2001a). We discussed this in Sections 5.1.4 and 4.1.3 respectively.
- **Automatic labeling, variable projection.** While progress has been made on several accounts of constraint solver support, these topics have not been addressed yet. The topic of projection, the elimination of existentially quantified variables, is particularly challenging to generalize to arbitrary CHR solvers. Any progress would be highly significant.
- **Partial evaluation.** Section 4.2.3 mentions recent work in this area (Frühwirth 2005c; Tacchella, Gabbrielli et al. 2007; Sarna-Starosta and Schrijvers 2007). For now, it is clear that the multi-headedness of CHR makes a straightforward application of partial evaluation techniques for conventional languages to CHR programs nearly impossible. Further investigation of techniques tailored towards CHR are necessary to make substantial improvements.
- **Abstract interpretation.** A general framework for abstract interpretation in the context of CHR was proposed (Schrijvers, Stuckey et al. 2005) (see Section 4.2.2). However, only a limited number of analyses have been formulated in the framework so far. Moreover, little information is present in a CHR program on its own. In order to make analysis results more accurate, we require an analysis framework that encompasses both CHR and its host language. Such a framework should prove to be beneficial to the accuracy of the analyses for both languages.

8.3 Grand Challenges

Much progress has been made in the last ten years and many of the open problems have been resolved. However, a few difficult questions are still unresolved, and in the meantime many more problems have become apparent.

In our view the following four topics are *grand challenges* that must be addressed by the CHR research community in the next decade. These four grand challenges are not only of technical interest, they are also vital for the further adoption of the CHR community and user-base.

1. **Programming environments and tools.** If measured by current standards, which dictate that a language is only as good as its tools, CHR is a poor language indeed. While several strong theoretical results have been obtained in the field of program analysis for CHR, little (if any) effort has been

made to embody these results into a practical tool for day-to-day programming. For example, programmers have to manually check for confluence, and, in the case of non-confluence, complete their solvers by hand.

2. **Execution control.** Compared to the refined operational semantics, the rule priorities of the CHR^{rp} semantics are a step in the realization of Kowalski's slogan "Algorithm = Logic + Control" in the context of CHR. However, there are still many challenges in finding satisfactory ways to allow programmers to fine-tune the execution strategy of their programs.

3. **Parallelism, concurrency.** Recent theoretical work (Frühwirth 2005b; Meister 2006) confirms CHR's inherent aptness for parallel programming. Truly leveraging the full power of current and future multi-core processors through CHR, however, requires practical, efficient, concurrent implementations. Currently, these implementations are still in early stages (cf. Section 4.1.2 for a discussion of some early Haskell-based prototypes). Many important problems are still to be researched in this domain, from language features and semantics, to analysis, implementation, and optimization.

4. **Scaling to industrial applications.** Strong theoretical results have been obtained concerning the performance of CHR (cf. Section 3.3.2), and these have also been reflected in the actual runtimes of CHR programs. However, CHR is still at least one or two orders of magnitude slower than most conventional programming languages and constraint solvers. This becomes particularly apparent for CHR applications that surpass the toy research programs of 10 lines: industrial applications for instance, such as those mentioned in Section 7.4, easily count 100 to 1000 lines. The refined semantics compilation scheme (see Section 4) was not designed or benchmarked with such program sizes in mind. Some potential scalability aspects are:

    - huge constraint stores that have to be persistent and/or distributed;
    - (dynamic) optimizations, also for variants and extensions of CHR;
    - incremental compilation, run-time rule assertion, reflection;
    - higher-order / meta-programming.

    These and other aspects must be investigated to achieve further industrial adoption.

Acknowledgments

References

ABDENNADHER, S. 2000. A language for experimenting with declarative paradigms. In *RCoRP '00(bis): Proc. 2nd Workshop on Rule-Based Constraint Reasoning and Programming*, T. Frühwirth et al., Eds.

ABDENNADHER, S. 2001. Rule-based constraint programming: Theory and practice. Habilitationsschrift. Institute of Computer Science, LMU, Munich, Germany.

ABDENNADHER, S. AND CHRISTIANSEN, H. 2000. An experimental CLP platform for integrity constraints and abduction. In *FQAS '00: Proc. 4th Intl. Conf. Flexible Query Answering Systems*. Springer, 141–152.

ABDENNADHER, S. AND FRÜHWIRTH, T. 1998. On completion of Constraint Handling Rules. In *CP '98*, M. J. Maher and J.-F. Puget, Eds. LNCS, vol. 1520. Springer, 25–39.

ABDENNADHER, S. AND FRÜHWIRTH, T. 1999. Operational equivalence of CHR programs and constraints. In *CP '99*, J. Jaffar, Ed. LNCS, vol. 1713. Springer, 43–57.

ABDENNADHER, S. AND FRÜHWIRTH, T. 2004. Integration and optimization of rule-based constraint solvers. In *LOPSTR '03*, M. Bruynooghe, Ed. LNCS, vol. 3018. Springer, 198–213.

ABDENNADHER, S., FRÜHWIRTH, T., AND HOLZBAUR, C., Eds. 2005. *Special Issue on Constraint Handling Rules*. Theory and Practice of Logic Programming, vol. 5(4–5).

ABDENNADHER, S., FRÜHWIRTH, T., AND MEUSS, H. 1999. Confluence and semantics of constraint simplification rules. *Constraints 4,* 2, 133–165.

ABDENNADHER, S., KRÄMER, E., SAFT, M., AND SCHMAUSS, M. 2002. JACK: A Java Constraint Kit. In *WFLP '01: Proc. 10th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, M. Hanus, Ed. ENTCS, vol. 64. Elsevier, 1–17. See also http://pms.ifi.lmu.de/software/jack/.

ABDENNADHER, S. AND MARTE, M. 2000. University course timetabling using Constraint Handling Rules. In Holzbaur and Frühwirth (2000b), 311–325.

ABDENNADHER, S., OLAMA, A., SALEM, N., AND THABET, A. 2006. ARM: Automatic Rule Miner. In *LOPSTR '06, Revised Selected Papers*. LNCS, vol. 4407. Springer.

ABDENNADHER, S. AND RIGOTTI, C. 2004. Automatic generation of rule-based constraint solvers over finite domains. *ACM TOCL 5,* 2, 177–205.

ABDENNADHER, S. AND RIGOTTI, C. 2005. Automatic generation of CHR constraint solvers. In Abdennadher, Frühwirth et al. (2005), 403–418.

ABDENNADHER, S. AND SAFT, M. 2001. A visualization tool for Constraint Handling Rules. In *WLPE '01*, A. Kusalik, Ed.

ABDENNADHER, S., SAFT, M., AND WILL, S. 2000. Classroom assignment using constraint logic programming. In *PACLP '00: Proc. 2nd Intl. Conf. and Exhibition on Practical Application of Constraint Technologies and Logic Programming*.

ABDENNADHER, S. AND SCHÜTZ, H. 1998. CHR$^\vee$, a flexible query language. In *FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems*, T. Andreasen, H. Christiansen, and H. Larsen, Eds. LNAI, vol. 1495. Springer, 1–14.

ABDENNADHER, S. AND SOBHI, I. 2008. Generation of rule-based constraint solvers: Combined approach. In *LOPSTR '07, Revised Selected Papers*, A. King, Ed. LNCS, vol. 4915.

AGUILAR-SOLIS, D. AND DAHL, V. 2004. Coordination revisited – a Constraint Handling Rule approach. In *IBERAMIA '04: Proc. 9th Ibero-American Conf. on AI*. LNCS, vol. 3315. 315–324.

ALBERTI, M., CHESANI, F., GAVANELLI, M., AND LAMMA, E. 2005. The CHR-based implementation of a system for generation and confirmation of hypotheses. In Wolf, Frühwirth et al. (2005), 111–122.

ALBERTI, M., DAOLIO, D., TORRONI, P., GAVANELLI, M., LAMMA, E., AND MELLO, P. 2004. Specification and verification of agent interaction protocols in a logic-based system. In *SAC '04: Proc. 19th ACM Symp. Applied Computing*, H. Haddad et al., Eds. ACM Press, 72–78.

ALBERTI, M., GAVANELLI, M., LAMMA, E., CHESANI, F., MELLO, P., AND TORRONI, P. 2006. Compliance verification of agent interaction: a logic-based software tool. *Applied Artificial Intelligence 20,* 2–4, 133–157.

ALBERTI, M., GAVANELLI, M., LAMMA, E., MELLO, P., AND MILANO, M. 2005. A CHR-based implementation of known arc-consistency. In Abdennadher, Frühwirth et al. (2005), 419–440.

ALBERTI, M., GAVANELLI, M., LAMMA, E., MELLO, P., AND TORRONI, P. 2004. Specification and verification of agent interaction using social integrity constraints. In *LCMAS'03: Logic and Communication in Multi-Agent Systems*. ENTCS, vol. 85(2). Elsevier, 94–116.

ALVES, S. AND FLORIDO, M. 2002. Type inference using Constraint Handling Rules. In *WFLP '01: Proc. 10th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, M. Hanus, Ed. ENTCS, vol. 64. Elsevier, 56–72.

APT, K. R. AND MONFROY, E. 2001. Constraint programming viewed as rule-based programming. *TPLP 1,* 6, 713–750.

BADEA, L., TILIVEA, D., AND HOTARAN, A. 2004. Semantic Web Reasoning for Ontology-Based Integration of Resources. *PPSWR '04: Proc. 2nd Intl. Workshop on Principles And Practice Of Semantic Web Reasoning 3208*, 61–75.

BARRANCO-MENDOZA, A. 2005. Stochastic and heuristic modelling for analysis of the growth of pre-invasive lesions and for a multidisciplinary approach to early cancer diagnosis. Ph.D. thesis, Simon Fraser University, Canada.

BAVARIAN, M. AND DAHL, V. 2006. Constraint based methods for biological sequence analysis. *J. UCS 12,* 11, 1500–1520.

BÈS, G. G. AND DAHL, V. 2003. Balanced parentheses in NL texts: a useful cue in the syntax/semantics interface. In *Proc. Lorraine-Saarland Workshop on Prospects and Advances in the Syntax/Semantics Interface*. Poster Paper.

BETZ, H. 2007. Relating coloured Petri nets to Constraint Handling Rules. In Djelloul, Duck et al. (2007), 33–47.

BETZ, H. AND FRÜHWIRTH, T. 2005. A linear-logic semantics for Constraint Handling Rules. In *CP '05*. LNCS, vol. 3709. Springer, 137–151.

BETZ, H. AND FRÜHWIRTH, T. 2007. A linear-logic semantics for Constraint Handling Rules with disjunction. In Djelloul, Duck et al. (2007), 17–31.

BISTARELLI, S., FRÜHWIRTH, T., MARTE, M., AND ROSSI, F. 2004. Soft constraint propagation and solving in Constraint Handling Rules. *Computational Intelligence: Special Issue on Preferences in AI and CP 20,* 2 (May), 287–307.

BOESPFLUG, M. 2007. TaiChi:how to check your types with serenity. *The Monad.Reader 9*, 17–31.

BOUISSOU, O. 2004. A CHR library for SiLCC. Diplomathesis. Technical University of Berlin, Germany.

BRAND, S. 2002. A note on redundant rules in rule-based constraint programming. In *Joint ERCIM/CologNet Intl. Workshop on Constraint Solving and Constraint Logic Programming, Selected papers*. LNCS, vol. 2627. Springer, 279–336.

BRAND, S. AND MONFROY, E. 2003. Deductive generation of constraint propagation rules. In *RULE '03: 4th Intl. Workshop on Rule-Based Programming*, G. Vidal, Ed. ENTCS, vol. 86(2). Elsevier, 45–60.

Bressan, S. and Goh, C. H. 1998. Answering queries in context. In *FQAS '98: Proc. 3rd Intl. Conf. on Flexible Query Answering Systems*, T. Andreasen, H. Christiansen, and H. Larsen, Eds. LNAI, vol. 1495. Springer, 68–82.

Cabedo, L. M. and Escrig, M. T. 2003. Modeling motion by the integration of topology and time. *J. UCS 9,* 9, 1096–1122.

Chin, W.-N., Craciun, F., Khoo, S.-C., and Popeea, C. 2006. A flow-based approach for variant parametric types. *SIGPLAN Not. 41,* 10, 273–290.

Chin, W.-N., Sulzmann, M., and Wang, M. 2003. A type-safe embedding of Constraint Handling Rules into Haskell. Honors Thesis. School of Computing, National University of Singapore.

Christiansen, H. 2005. CHR grammars. In Abdennadher, Frühwirth et al. (2005), 467–501.

Christiansen, H. 2006. On the implementation of global abduction. In *CLIMA '06: 7th Intl. Workshop on Computational Logic in Multi-Agent Systems – Revised, Selected and Invited Papers*, K. Inoue, K. Satoh, and F. Toni, Eds. LNCS, vol. 4371. Springer, 226–245.

Christiansen, H. and Dahl, V. 2003. Logic grammars for diagnosis and repair. *Intl. J. Artificial Intelligence Tools 12,* 3, 227–248.

Christiansen, H. and Dahl, V. 2005a. HYPROLOG: A new logic programming language with assumptions and abduction. In Gabbrielli and Gupta (2005), 159–173.

Christiansen, H. and Dahl, V. 2005b. Meaning in context. In *CONTEXT '05: Proc. 4th Intl. and Interdisciplinary Conf. Modeling and Using Context*. LNAI, vol. 3554. Springer, 97–111.

Christiansen, H. and Have, C. T. 2007. From use cases to UML class diagrams using logic grammars and constraints. In *RANLP '07: Proc. Intl. Conf. Recent Adv. Nat. Lang. Processing*. 128–132.

Christiansen, H. and Martinenghi, D. 2000. Symbolic constraints for meta-logic programming. In Holzbaur and Frühwirth (2000b), 345–367.

Coquery, E. and Fages, F. 2003. TCLP: A type checker for CLP($\mathcal{X}$). In *WLPE '03*, F. Mesnard and A. Serebrenik, Eds. K.U.Leuven, Dept. Comp. Sc., Technical report CW 371. 17–30.

Coquery, E. and Fages, F. 2005. A type system for CHR. In Schrijvers and Frühwirth (2005b), 19–33.

Dahl, V. 2004. An abductive treatment of long distance dependencies in CHR. In *CSLP '04: Proc. First Intl. Workshop on Constraint Solving and Language Processing*. LNCS, vol. 3438. Springer, 17–31. Invited Paper.

Dahl, V. and Blache, P. 2005. Extracting selected phrases through constraint satisfaction. In *Proc. 2nd Intl. Workshop on Constraint Solving and Language Processing*.

Dahl, V. and Gu, B. 2006. Semantic property grammars for knowledge extraction from biomedical text. In Etalle and Truszczynski (2006), 442–443. Poster Paper.

Dahl, V. and Gu, B. 2007. A CHRG analysis of ambiguity in biological texts. In *CSLP '07: Proc. 4th Intl. Workshop on Constraints and Language Processing*. Extended Abstract.

Dahl, V. and Niemelä, I., Eds. 2007. *ICLP '07: Proc. 23rd Intl. Conf. Logic Programming*. LNCS, vol. 4670. Springer.

Dahl, V. and Voll, K. 2004. Concept formation rules: An executable cognitive model of knowledge construction. In *NLUCS '04: Proc. First Intl. Workshop on Natural Language Understanding and Cognitive Sciences*.

De Koninck, L., Schrijvers, T., and Demoen, B. 2006a. INCLP($\mathbb{R}$) - Interval-based

nonlinear constraint logic programming over the reals. In Fink, Tompits et al. (2006), 91–100.

DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2006b. Search strategies in CHR(Prolog). In Schrijvers and Frühwirth (2006), 109–123.

DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2007a. The correspondence between the Logical Algorithms language and CHR. In Dahl and Niemelä (2007), 209–223.

DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2007b. User-definable rule priorities for CHR. In PPDP '07, M. Leuschel and A. Podelski, Eds. ACM Press, 25–36.

DE KONINCK, L. AND SNEYERS, J. 2007. Join ordering for Constraint Handling Rules. In Djelloul, Duck et al. (2007), 107–121.

DE KONINCK, L., STUCKEY, P. J., AND DUCK, G. J. 2008. Optimizing compilation of CHR with rule priorities. In Proc. 9th Intl. Symp. Functional and Logic Programming, J. Garrigue and M. Hermenegildo, Eds. LNCS, vol. 4989. Springer, 32–47.

DELZANNO, G., GABBRIELLI, M., AND MEO, M. C. 2005. A compositional semantics for CHR. In PPDP '05, P. Barahona and A. Felty, Eds. ACM Press, 209–217.

DEMOEN, B. AND LIFSCHITZ, V., Eds. 2004. ICLP '04: Proc. 20th Intl. Conf. Logic Programming. LNCS, vol. 3132. Springer.

DJELLOUL, K., DAO, T.-B.-H., AND FRÜHWIRTH, T. 2007. Toward a first-order extension of Prolog's unification using CHR: a CHR first-order constraint solver over finite or infinite trees. In SAC '07: Proc. 2007 ACM Symp. Applied computing. ACM Press, 58–64.

DJELLOUL, K., DUCK, G. J., AND SULZMANN, M., Eds. 2007. CHR '07: Proc. 4th Workshop on Constraint Handling Rules.

DUCASSÉ, M. 1999. Opium: an extendable trace analyzer for Prolog. J. Logic Programming 39, 1–3, 177–223.

DUCK, G. J. 2005. Compilation of Constraint Handling Rules. Ph.D. thesis, University of Melbourne, Victoria, Australia.

DUCK, G. J. AND SCHRIJVERS, T. 2005. Accurate functional dependency analysis for Constraint Handling Rules. In Schrijvers and Frühwirth (2005b), 109–124.

DUCK, G. J., STUCKEY, P. J., AND BRAND, S. 2006. ACD term rewriting. In Etalle and Truszczynski (2006), 117–131.

DUCK, G. J., STUCKEY, P. J., GARCÍA DE LA BANDA, M., AND HOLZBAUR, C. 2003. Extending arbitrary solvers with Constraint Handling Rules. In PPDP '03. ACM Press, 79–90.

DUCK, G. J., STUCKEY, P. J., GARCÍA DE LA BANDA, M., AND HOLZBAUR, C. 2004. The refined operational semantics of Constraint Handling Rules. In Demoen and Lifschitz (2004), 90–104.

DUCK, G. J., STUCKEY, P. J., AND SULZMANN, M. 2007. Observable confluence for Constraint Handling Rules. In Dahl and Niemelä (2007), 224–239.

ESCRIG, M. T. AND TOLEDO, F. 1998a. A framework based on CLP extended with CHRs for reasoning with qualitative orientation and positional information. J. Visual Languages and Computing 9, 1, 81–101.

ESCRIG, M. T. AND TOLEDO, F. 1998b. Qualitative Spatial Reasoning: Theory and Practice — Application to Robot Navigation. IOS Press.

ETALLE, S. AND TRUSZCZYNSKI, M., Eds. 2006. ICLP '06: Proc. 22nd Intl. Conf. Logic Programming. LNCS, vol. 4079. Springer.

FINK, M., TOMPITS, H., AND WOLTRAN, S., Eds. 2006. WLP '06: Proc. 20th Workshop on Logic Programming. T.U.Wien, Austria, INFSYS Research report 1843-06-02.

FIRAT, A. 2003. Information integration using contextual knowledge and ontology merging. Ph.D. thesis, MIT Sloan School of Management, Cambridge, Massachusetts, USA.

FRÜHWIRTH, T. 1992. Constraint simplification rules. Tech. Rep. ECRC-92-18, European Computer-Industry Research Centre, Munich, Germany. July.

FRÜHWIRTH, T. 1995. Constraint Handling Rules. In *Constraint Programming: Basic and Trends — Selected Papers of the 22nd Spring School in Theoretical Computer Sciences, May 16–20, 1994*, A. Podelski, Ed. LNCS, vol. 910. Springer, 90–107.

FRÜHWIRTH, T. 1998. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming 37,* 1–3, 95–138.

FRÜHWIRTH, T. 2000. Proving termination of constraint solver programs. In *New Trends in Constraints, Joint ERCIM/Compulog Net Workshop, October 1999, Selected papers*, K. Apt, A. Kakas, E. Monfroy, and F. Rossi, Eds. LNCS, vol. 1865. Springer, 298–317.

FRÜHWIRTH, T. 2001. On the number of rule applications in constraint programs. In *Declarative Programming - Selected Papers from AGP 2000*, A. Dovier, M. C. Meo, and A. Omicini, Eds. ENTCS, vol. 48. Elsevier, 147–166.

FRÜHWIRTH, T. 2002a. As time goes by: Automatic complexity analysis of simplification rules. In *KR '02: Proc. 8th Intl. Conf. Princ. Knowledge Representation and Reasoning*, D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, Eds. Morgan Kaufmann, 547–557.

FRÜHWIRTH, T. 2002b. As time goes by II: More automatic complexity analysis of concurrent rule programs. In *QAPL '01: Proc. First Intl. Workshop on Quantitative Aspects of Programming Languages*, A. D. Pierro and H. Wiklicky, Eds. ENTCS, vol. 59(3). Elsevier.

FRÜHWIRTH, T. 2005a. Logical rules for a lexicographic order constraint solver. In Schrijvers and Frühwirth (2005b), 79–91.

FRÜHWIRTH, T. 2005b. Parallelizing union-find in Constraint Handling Rules using confluence. In Gabbrielli and Gupta (2005), 113–127.

FRÜHWIRTH, T. 2005c. Specialization of concurrent guarded multi-set transformation rules. In *LOPSTR '04*, S. Etalle, Ed. LNCS, vol. 3573. Springer, 133–148.

FRÜHWIRTH, T. 2006a. Complete propagation rules for lexicographic order constraints over arbitrary domains. In *Recent Advances in Constraints, CSCLP '05: Joint ERCIM/CoLogNET Intl. Workshop on Constraint Solving and CLP, Revised Selected and Invited Papers*. LNAI, vol. 3978. Springer.

FRÜHWIRTH, T. 2006b. Deriving linear-time algorithms from union-find in CHR. In Schrijvers and Frühwirth (2006), 49–60.

FRÜHWIRTH, T. 2007. Description logic and rules the CHR way. In Djelloul, Duck et al. (2007), 49–61.

FRÜHWIRTH, T. 2009. *Constraint Handling Rules*. Cambridge University Press. To appear.

FRÜHWIRTH, T. ET AL., Eds. 2000. *RCoRP '00: Proc. 1st Workshop on Rule-Based Constraint Reasoning and Programming*.

FRÜHWIRTH, T. AND ABDENNADHER, S. 2001. The Munich rent advisor: A success for logic programming on the internet. *TPLP 1,* 3, 303–319.

FRÜHWIRTH, T. AND ABDENNADHER, S. 2003. *Essentials of Constraint Programming*. Springer.

FRÜHWIRTH, T. AND BRISSET, P. 1995. High-level implementations of Constraint Handling Rules. Tech. Rep. ECRC-95-20, European Computer-Industry Research Centre.

FRÜHWIRTH, T. AND BRISSET, P. 1998. Optimal placement of base stations in wireless indoor telecommunication. In *CP '98*, M. J. Maher and J.-F. Puget, Eds. LNCS, vol. 1520. Springer, 476–480.

FRÜHWIRTH, T. AND BRISSET, P. 2000. Placing base stations in wireless indoor communication networks. *IEEE Intelligent Systems and Their Applications 15,* 1, 49–53.

FRÜHWIRTH, T., DI PIERRO, A., AND WIKLICKY, H. 2002. Probabilistic Constraint Handling Rules. In *WFLP '02: Proc. 11th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers*, M. Comini and M. Falaschi, Eds. ENTCS, vol. 76. Elsevier.

FRÜHWIRTH, T. AND HOLZBAUR, C. 2003. Source-to-source transformation for a class of expressive rules. In *AGP '03: Joint Conf. Declarative Programming APPIA-GULP-PRODE*, F. Buccafurri, Ed. 386–397.

FRÜHWIRTH, T. AND MEISTER, M., Eds. 2004. *CHR '04: 1st Workshop on Constraint Handling Rules: Selected Contributions.*

GABBRIELLI, M. AND GUPTA, G., Eds. 2005. *ICLP '05: Proc. 21st Intl. Conf. Logic Programming.* LNCS, vol. 3668. Springer.

GABBRIELLI, M. AND MEO, M. C. 2009. A compositional semantics for CHR. *ACM TOCL 10,* 2.

GANZINGER, H. AND MCALLESTER, D. A. 2002. Logical algorithms. In Stuckey (2002), 209–223.

GARAT, D. AND WONSEVER, D. 2002. A constraint parser for contextual rules. In *Proc. 22nd Intl. Conf. of the Chilean Computer Science Society*. IEEE Computer Society, 234–242.

GAVANELLI, M., LAMMA, E., MELLO, P., ET AL. 2003. Interpreting abduction in CLP. In *AGP '03: Joint Conf. Declarative Programming APPIA-GULP-PRODE*, F. Buccafurri, Ed. 25–35.

GEURTS, J., VAN OSSENBRUGGEN, J., AND HARDMAN, L. 2001. Application-specific constraints for multimedia presentation generation. In *MMM '01: Proc. 8th Intl. Conf. on Multimedia Modeling.* 247–266.

GOURAUD, S.-D. AND GOTLIEB, A. 2006. Using CHRs to generate functional test cases for the Java card virtual machine. In *PADL '06: Proc. 8th Intl. Symp. Practical Aspects of Declarative Languages*, P. Van Hentenryck, Ed. LNCS, vol. 3819. Springer, 1–15.

HAEMMERLÉ, R. AND FAGES, F. 2007. Abstract critical pairs and confluence of arbitrary binary relations. In *RTA '07: Proc. 18th Intl. Conf. Term Rewriting and Applications.* LNCS, vol. 4533. Springer.

HANUS, M. 2006. Adding Constraint Handling Rules to Curry. In Fink, Tompits et al. (2006), 81–90.

HECKSHER, T., NIELSEN, S. T., AND PIGEON, A. 2002. A CHRG model of the ancient Egyptian grammar. Unpublished student project report, Roskilde University, Denmark.

HOLZBAUR, C. AND FRÜHWIRTH, T. 1998. Constraint Handling Rules reference manual, release 2.2. Tech. Rep. TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, Wien.

HOLZBAUR, C. AND FRÜHWIRTH, T. 1999. Compiling Constraint Handling Rules into Prolog with attributed variables. In *PPDP '99*, G. Nadathur, Ed. LNCS, vol. 1702. Springer, 117–133.

HOLZBAUR, C. AND FRÜHWIRTH, T. 2000a. A Prolog Constraint Handling Rules compiler and runtime system. In Holzbaur and Frühwirth (2000b), 369–388.

HOLZBAUR, C. AND FRÜHWIRTH, T., Eds. 2000b. *Special Issue on Constraint Handling Rules.* Journal of Applied Artificial Intelligence, vol. 14(4).

HOLZBAUR, C., GARCÍA DE LA BANDA, M., STUCKEY, P. J., AND DUCK, G. J. 2005. Optimizing compilation of Constraint Handling Rules in HAL. In Abdennadher, Frühwirth et al. (2005), 503–531.

KÄSER, M. AND MEISTER, M. 2006. Implementation of an F-Logic kernel in CHR. In Schrijvers and Frühwirth (2006), 33–47.

KOSMATOV, N. 2006a. A constraint solver for sequences and its applications. In *Proc. 2006 ACM Symp. on Applied Computing*. ACM Press, 404–408.

KOSMATOV, N. 2006b. Constraint solving for sequences in software validation and verification. In *INAP '05: Proc. 16th Intl. Conf. Applications of Declarative Programming and Knowledge Management*. LNCS, vol. 4369. Springer, 25–37.

KRÄMER, E. 2001. A generic search engine for a Java Constraint Kit. Diplomarbeit. Institute of Computer Science, LMU, Munich, Germany.

LAM, E. S. AND SULZMANN, M. 2006. Towards agent programming in CHR. In Schrijvers and Frühwirth (2006), 17–31.

LAM, E. S. AND SULZMANN, M. 2007. A concurrent Constraint Handling Rules semantics and its implementation with software transactional memory. In *DAMP '07: Proc. ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. ACM Press. System's homepage at `http://taichi.ddns.comp.nus.edu.sg/taichiwiki/CCHR/`.

LÖTZBEYER, H. AND PRETSCHNER, A. 2000. AUTOFOCUS on constraint logic programming. In *LPSE '00: Proc. Intl. Workshop on (Constraint) Logic Programming and Software Engineering*.

MAHER, M. J. 2002. Propagation completeness of reactive constraints. In Stuckey (2002), 148–162.

MEISTER, M. 2006. Fine-grained parallel implementation of the preflow-push algorithm in CHR. In Fink, Tompits et al. (2006), 172–181.

MEISTER, M., DJELLOUL, K., AND FRÜHWIRTH, T. 2006. Complexity of a CHR solver for existentially quantified conjunctions of equations over trees. In *CSCLP '06: Proc. 11th Annual ERCIM Workshop on Constraint Solving and Constraint Programming*, F. Azevedo et al., Eds. LNCS, vol. 4651. Springer, 139–153.

MEISTER, M., DJELLOUL, K., AND ROBIN, J. 2007. A unified semantics for Constraint Handling Rules in transaction logic. In *LPNMR '07: Proc. 9th Intl. Conf. Logic Programming and Nonmonotonic Reasoning*, C. Baral, G. Brewka, and J. S. Schlipf, Eds. LNCS, vol. 4483. Springer, 201–213.

MENEZES, L., VITORINO, J., AND AURELIO, M. 2005. A high performance $CHR^{\vee}$ execution engine. In Schrijvers and Frühwirth (2005b), 35–45.

MEYER, B. 2000. A constraint-based framework for diagrammatic reasoning. In Holzbaur and Frühwirth (2000b), 327–344.

MORAWIETZ, F. 2000. Chart parsing and constraint programming. In *COLING '00: Proc. 18th Intl. Conf. on Computational Linguistics*, M. Kay, Ed. Morgan Kaufmann.

MORAWIETZ, F. AND BLACHE, P. 2002. Parsing natural languages with CHR. Unpublished Draft.

PENN, G. 2000. Applying Constraint Handling Rules to HPSG. In Frühwirth et al. (2000).

PILOZZI, P., SCHRIJVERS, T., AND DE SCHREYE, D. 2007. Proving termination of CHR in Prolog: A transformational approach. In *WST '07: 9th Intl. Workshop on Termination*.

PRETSCHNER, A., SLOTOSCH, O., AIGLSTORFER, E., AND KRIEBEL, S. 2004. Model-based testing for real. *J. Software Tools for Technology Transfer (STTT) 5,* 2–3, 140–157.

RAISER, F. 2007. Graph transformation systems in CHR. In Dahl and Niemelä (2007), 240–254.

RAISER, F. AND TACCHELLA, P. 2007. On confluence of non-terminating CHR programs. In Djelloul, Duck et al. (2007), 63–76.

RIBEIRO, C., ZÚQUETE, A., FERREIRA, P., AND GUEDES, P. 2000. Security policy consistency. In Frühwirth et al. (2000).

RINGWELSKI, G. AND SCHLENKER, H. 2000a. Type inference in CHR programs for the composition of constraint systems. In *WLP '00: Proc. 15th Workshop on Logic Programming*, S. Abdennadher, U. Geske, and D. Seipel, Eds. 137–146.

RINGWELSKI, G. AND SCHLENKER, H. 2000b. Using typed interfaces to compose CHR programs. In *RCoRP '00(bis): Proc. 2nd Workshop on Rule-Based Constraint Reasoning and Programming*, T. Frühwirth et al., Eds.

ROBIN, J. AND VITORINO, J. 2006. ORCAS: Towards a CHR-based model-driven framework of reusable reasoning components. In Fink, Tompits et al. (2006), 192–199.

ROBIN, J., VITORINO, J., AND WOLF, A. 2007. Constraint programming architectures: Review and a new proposal. *J. UCS 13,* 6, 701–720.

SARNA-STAROSTA, B. AND RAMAKRISHNAN, C. 2007. Compiling Constraint Handling Rules for efficient tabled evaluation. In *PADL '07: Proc. 9th Intl. Symp. Practical Aspects of Declarative Languages*, M. Hanus, Ed. LNCS, vol. 4354. Springer, 170–184. System's homepage at `http://www.cse.msu.edu/~bss/chr_d`.

SARNA-STAROSTA, B. AND SCHRIJVERS, T. 2007. Indexing techniques for CHR based on program transformation. Tech. Rep. CW 500, K.U.Leuven, Dept. Comp. Sc. Aug.

SCHIFFEL, S. AND THIELSCHER, M. 2007. Fluxplayer: A successful general game player. In *AAAI '07: Proc. 22nd AAAI Conf. Artificial Intelligence*. AAAI Press, 1191–1196.

SCHMAUSS, M. 1999. An implementation of CHR in Java. Diplomarbeit. Institute of Computer Science, LMU, Munich, Germany.

SCHRIJVERS, T. 2004. Jmmsolve: A generative java memory model implemented in Prolog and CHR. In Demoen and Lifschitz (2004), 475–476.

SCHRIJVERS, T. 2005. Analyses, optimizations and extensions of Constraint Handling Rules. Ph.D. thesis, K.U.Leuven, Leuven, Belgium.

SCHRIJVERS, T. AND BRUYNOOGHE, M. 2006. Polymorphic algebraic data type reconstruction. In *PPDP '06*, A. Bossi and M. Maher, Eds. ACM Press, 85–96.

SCHRIJVERS, T. AND DEMOEN, B. 2004a. Antimonotony-based delay avoidance for CHR. Tech. Rep. CW 385, K.U.Leuven, Dept. Comp. Sc. July.

SCHRIJVERS, T. AND DEMOEN, B. 2004b. The K.U.Leuven CHR system: Implementation and application. In Frühwirth and Meister (2004), 8–12. System's homepage at `http://www.cs.kuleuven.be/~toms/CHR/`.

SCHRIJVERS, T., DEMOEN, B., DUCK, G. J., STUCKEY, P. J., AND FRÜHWIRTH, T. 2006. Automatic implication checking for CHR constraints. In *RULE '05: 6th Intl. Workshop on Rule-Based Programming*. ENTCS, vol. 147(1). Elsevier, 93–111.

SCHRIJVERS, T. AND FRÜHWIRTH, T. 2005a. Analysing the CHR implementation of union-find. In Wolf, Frühwirth et al. (2005), 135–146.

SCHRIJVERS, T. AND FRÜHWIRTH, T., Eds. 2005b. *CHR '05: Proc. 2nd Workshop on Constraint Handling Rules*. K.U.Leuven, Dept. Comp. Sc., Technical report CW 421.

SCHRIJVERS, T. AND FRÜHWIRTH, T., Eds. 2006. *CHR '06: Proc. 3rd Workshop on Constraint Handling Rules*. K.U.Leuven, Dept. Comp. Sc., Technical report CW 452.

SCHRIJVERS, T. AND FRÜHWIRTH, T. 2006. Optimal union-find in Constraint Handling Rules. *TPLP 6,* 1–2, 213–224.

SCHRIJVERS, T., STUCKEY, P. J., AND DUCK, G. J. 2005. Abstract interpretation for Constraint Handling Rules. In *PPDP '05*, P. Barahona and A. Felty, Eds. ACM Press, 218–229.

SCHRIJVERS, T. AND WARREN, D. S. 2004. Constraint Handling Rules and tabled execution. In Demoen and Lifschitz (2004), 120–136.

SCHRIJVERS, T., WARREN, D. S., AND DEMOEN, B. 2003. CHR for XSB. In *CICLOPS '03: Proc. 3rd Intl. Colloq. on Implementation of Constraint and Logic Programming Systems*, R. Lopes and M. Ferreira, Eds. University of Porto, Portugal, Dept. Comp. Sc., Technical report DCC-2003-05. 7–20.

SCHRIJVERS, T., WIELEMAKER, J., AND DEMOEN, B. 2005. Poster: Constraint Handling Rules for SWI-Prolog. In Wolf, Frühwirth et al. (2005).

Schrijvers, T., Zhou, N.-F., and Demoen, B. 2006. Translating Constraint Handling Rules into Action Rules. In Schrijvers and Frühwirth (2006), 141–155.

Schumann, E. T. 2002. A literate programming system for logic programs with constraints. In *WFLP '02: Proc. 11th Intl. Workshop on Functional and (Constraint) Logic Programming*, M. Comini and M. Falaschi, Eds. University of Udine, Research Report UDMI/18/2002/RR.

Scientific Software & Systems Ltd. 2008. Company Profile: Solving problems with proven solutions. Available at `http://www.sss.co.nz/`.

Seitz, C., Bauer, B., and Berger, M. 2002. Planning and scheduling in multi agent systems using Constraint Handling Rules. In *IC-AI '02: Proc. Intl. Conf. Artificial Intelligence*. CSREA Press.

Shigeta, Y., Akama, K., Mabuchi, H., and Koike, H. 2006. Converting Constraint Handling Rules to Equivalent Transformation Rules. *JACIII 10,* 3, 339–348.

Simões, H. and Florido, M. 2004. TypeTool: A type inference visualization tool. In *WFLP '04: Proc. 13th Intl. Workshop on Functional and (Constraint) Logic Programming*, H. Kuchen, Ed. RWTH Aachen, Dept. Comp. Sc., Technical report AIB-2004-05. 48–61.

Sneyers, J., Schrijvers, T., and Demoen, B. 2005. Guard and continuation optimization for occurrence representations of CHR. In Gabbrielli and Gupta (2005), 83–97.

Sneyers, J., Schrijvers, T., and Demoen, B. 2006a. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In Fink, Tompits et al. (2006), 182–191.

Sneyers, J., Schrijvers, T., and Demoen, B. 2006b. Memory reuse for CHR. In Etalle and Truszczynski (2006), 72–86.

Sneyers, J., Schrijvers, T., and Demoen, B. 2009. The computational power and complexity of Constraint Handling Rules. *ACM TOPLAS 31,* 2.

Sneyers, J., Van Weert, P., and Schrijvers, T. 2007. Aggregates for Constraint Handling Rules. In Djelloul, Duck et al. (2007), 91–105.

Stuckey, P. J., Ed. 2002. *ICLP '02: Proc. 18th Intl. Conf. Logic Programming.* LNCS, vol. 2401. Springer.

Stuckey, P. J. and Sulzmann, M. 2005. A theory of overloading. *ACM TOPLAS 27,* 6, 1216–1269.

Sulzmann, M., Duck, G. J., Peyton-Jones, S., and Stuckey, P. J. 2007. Understanding functional dependencies via Constraint Handling Rules. *J. Functional Prog. 17,* 1, 83–129.

Sulzmann, M. and Lam, E. S. 2007a. Compiling Constraint Handling Rules with lazy and concurrent search techniques. In Djelloul, Duck et al. (2007), 139–149.

Sulzmann, M. and Lam, E. S. 2007b. Haskell - Join - Rules. In *IFL '07: 19th Intl. Symp. Implementation and Application of Functional Languages*, O. Chitil, Ed. 195–210.

Sulzmann, M., Schrijvers, T., and Stuckey, P. J. 2006. Principal type inference for GHC-style multi-parameter type classes. In *APLAS '06: Proc. 4th Asian Symp. on Programming Languages and Systems*, N. Kobayashi, Ed. LNCS, vol. 4279. Springer, 26–43.

Sulzmann, M., Wazny, J., and Stuckey, P. J. 2005. Constraint abduction and constraint handling rules. In Schrijvers and Frühwirth (2005b), 63–78.

Tacchella, P., Gabbrielli, M., and Meo, M. C. 2007. Unfolding in CHR. In *PPDP '07*, M. Leuschel and A. Podelski, Eds. ACM Press, 179–186.

Thielscher, M. 2002. Reasoning about actions with CHRs and finite domain constraints. In Stuckey (2002), 70–84.

THIELSCHER, M. 2005. FLUX: A logic programming method for reasoning agents. In Abdennadher, Frühwirth et al. (2005), 533–565.

UEDA, K. ET AL. 2006. LMNtal as a unifying declarative language. In Schrijvers and Frühwirth (2006), 1–15. Invited talk.

VAN WEERT, P. 2008. Compiling Constraint Handling Rules to Java: A reconstruction. Tech. Rep. CW 521, K.U.Leuven, Dept. Comp. Sc. Aug.

VAN WEERT, P., SCHRIJVERS, T., AND DEMOEN, B. 2005. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Schrijvers and Frühwirth (2005b), 47–62. System's homepage at `http://www.cs.kuleuven.be/∼petervw/JCHR/`.

VAN WEERT, P., SNEYERS, J., AND DEMOEN, B. 2008. Aggregates for CHR through program transformation. In *LOPSTR '07, Revised Selected Papers*, A. King, Ed. LNCS, vol. 4915.

VAN WEERT, P., SNEYERS, J., SCHRIJVERS, T., AND DEMOEN, B. 2006. Extending CHR with negation as absence. In Schrijvers and Frühwirth (2006), 125–140.

VAN WEERT, P., WUILLE, P., SCHRIJVERS, T., AND DEMOEN, B. 2008. CHR for imperative host languages. In *Special Issue on Constraint Handling Rules*. LNAI, vol. 5388. Springer.

VOETS, D., PILOZZI, P., AND DE SCHREYE, D. 2007. A new approach to termination analysis of Constraint Handling Rules. In Djelloul, Duck et al. (2007), 77–89.

VOLL, K. 2006. A methodology of error detection: Improving speech recognition in radiology. Ph.D. thesis, Simon Fraser University, Canada.

WAZNY, J. 2006. Type inference and type error diagnosis for Hindley/Milner with extensions. Ph.D. thesis, University of Melbourne, Australia.

WOLF, A. 1999. Adaptive Constraintverarbeitung mit Constraint-Handling-Rules – ein allgemeiner Ansatz zur Lösung dynamischer Constraint-probleme. Ph.D. thesis, Technical University Berlin, Berlin, Germany.

WOLF, A. 2000a. Projection in adaptive constraint handling. In *New Trends in Constraints, Joint ERCIM/Compulog Net Workshop, October 1999, Selected papers*, K. Apt, A. Kakas, E. Monfroy, and F. Rossi, Eds. LNCS, vol. 1865. Springer, 318–338.

WOLF, A. 2000b. Toward a rule-based solution of dynamic constraint hierarchies over finite domains. In Frühwirth et al. (2000).

WOLF, A. 2001a. Adaptive constraint handling with CHR in Java. In *CP '01*, T. Walsh, Ed. LNCS, vol. 2239. Springer, 256–270.

WOLF, A. 2001b. Attributed variables for dynamic constraint solving. In *Proc. 14th Intl. Conf. Applications of Prolog*. Prolog Association of Japan, 211–219.

WOLF, A. 2005. Intelligent search strategies based on adaptive Constraint Handling Rules. In Abdennadher, Frühwirth et al. (2005), 567–594.

WOLF, A., FRÜHWIRTH, T., AND MEISTER, M., Eds. 2005. *W(C)LP '05: Proc. 19th Workshop on (Constraint) Logic Programming*. Ulmer Informatik-Berichte, vol. 2005-01. Universität Ulm, Germany.

WOLF, A., GRUENHAGEN, T., AND GESKE, U. 2000. On incremental adaptation of CHR derivations. In Holzbaur and Frühwirth (2000b), 389–416.

WOLF, A., ROBIN, J., AND VITORINO, J. 2007. Adaptive CHR meets CHR$^\vee$: An extended refined operational semantics for CHR$^\vee$ based on justifications. In Djelloul, Duck et al. (2007), 1–15.

WUILLE, P., SCHRIJVERS, T., AND DEMOEN, B. 2007. CCHR: the fastest CHR implementation, in C. In Djelloul, Duck et al. (2007), 123–137. System's homepage at `http://www.cs.kuleuven.be/∼pieterw/CCHR/`.