Rule-Based Constraint Programming: Theory and Practice

Habilitationsschrift

Slim Abdennadher
Ludwig-Maximilians-Universität München
Institut für Informatik
Oettingenstr 67, D-80538 München

July 15, 2001

Gewidmet meiner Frau Nabila in Liebe.

Abstract

Constraint-based programming languages enjoy both elegant theoretical properties and practical success. As it runs, a constraint-based program successively generates pieces of partial information called constraints. The constraint solver has the task to collect, combine, and simplify the constraints, and detect their inconsistency. Intuitively, constraints represent elementary relationships between variables and values, for example equality or some order relationships. Clearly, the abilities and quality of the constraint solver play an essential role in constraint-based programming.

In the beginning, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language, termed the "black-box" approach. While efficient, this approach makes it hard to modify a solver or build a solver over a new domain, let alone reason about and analyze it. As the behavior of the solver can neither be inspected by the user nor explained by the computer, debugging of constraint-based programs is hard. Also, one lesson learned from practical applications is that constraints are often heterogeneous and application specific.

Several proposals have been made to allow more flexibility and customization of constraint solvers, often termed "glass-box" approaches. The most far-reaching proposal is the rule-based formalism. In this approach, the constraint propagation is achieved by repeated application of rules.

This work aims at covering all aspects of rule-based constraint programming. Going from theory to practice, we will present some analysis and debugging methods for rule-based constraint solvers. Then, we will present a new research area, where users for whom writing constraint solvers remains a hard task have the possibility to do it automatically. The paper then describes an implementation of a Java Constraint Kit consisting of a rule-based language for writing constraint solvers and a generic search engine to solve combinatorial problems. Then, we show how to extend a specific rule-based constraint language to become a general purpose language without losing the extra features supporting efficient constraint solving, e.g. committed choice and matching. Finally, we introduce two applications that benefit from using a rule-based constraint language for writing constraint solvers.

Acknowledgment

I would like to express my warmest gratitude to François Bry for his constant encouragement, for many productive discussions, and for his general support during all these years. I am also very grateful to Thom Frühwirth for the close and fruitful cooperation during the last six years.

Parts of the research presented in this thesis are based on work several students did under my supervision as part of their master's thesis or *Projektarbeit* at the Ludwig-Maximilians University: Matthias Schmauss implemented JCHR (Constraint Handling Rules in Java). Ekkerhard Krämer provided the search engine JASE for JCHR. Matthias Saft implemented the visualization tool for JCHR. Michael Marte implemented the timetabling system for the department of computer science. Without the practical work of all these people, this thesis would never have taken its present form.

I would like to thank my collaborators: Christophe Rigotti, Henning Christiansen (visitors to LMU), Thom Frühwirth, Holger Meuss, Heribert Schütz, Michael Marte (colleagues at LMU). Thanks also to Norbert Eisinger for his proofreading of various drafts of this thesis. The cooperation with Christophe Rigotti was partially supported by the Bavarian-French Hochschulzentrum.

I am also very grateful to François Bry, Martin Wirsing and François Fages for acting as referees of this work.

Last but not least, I thank my wife Nabila and my daughters Shirine and Amira for their ongoing support and patience.

Contents

Chapter 1

Introduction

Background

Constraint-based programming languages, be it constraint logic programming (CLP) [55, 98, 48, 70] or concurrent constraint logic programming [68, 87, 86, 56], enjoy both elegant theoretical properties and practical success. As it runs, a constraint-based program successively generates pieces of partial information called constraints. The constraint solver has the task to collect, combine, and simplify the constraints, and detect their inconsistency. Intuitively, constraints represent elementary relationships between variables and values, for example equality or some order relationships. Clearly, the abilities and quality of the constraint solver play an essential role in constraint-based programming.

In the beginning, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language, termed the "black-box" approach. While efficient, this approach makes it hard to modify a solver or build a solver over a new domain, let alone reason about and analyze it. As the behavior of the solver can neither be inspected by the user nor explained by the computer, debugging of constraint-based programs is hard. Also, one lesson learned from practical applications is that constraints are often heterogeneous and application specific.

Several proposals have been made to allow more flexibility and customization of constraint solvers, often termed "glass-box" approaches [36, 98]. The most-far reaching proposal is the rule-based formalism. In this approach, the constraint propagation is achieved by repeated application of rules.

Rule-based Constraint Programming

Rule-based programming began with Artificial Intelligence rule-based systems in the seventies. It is a powerful and elegant programming technique which often leads to very concise solutions that are very natural to write down. Rule-based formalisms become ubiquitous in computer science, and even more so in constraint reasoning and programming. In constraint reasoning, algorithms are often specified using inference rules, rewrite rules, sequents, or first-order axioms written as implications. It is no wonder that recently there has been a revival of interest in rule-based programming in the context of constraint programming. Advanced programming languages, like ELAN [59],CLAIRE [29], and Constraint Handling Rules (CHR) [46], have shown that the concept of rules could be of major interest as a programming tool for constraint solvers.

- ELAN is an environment for specifying and prototyping deduction systems, e.g. constraint solvers and theorem provers. It also provides a framework for experimenting with their combination. The ELAN system is based on labelled conditional rewrite systems and on strategies for controlling their application.
- CLAIRE is an object oriented programming language with rule processing capabilities. The goal of the rule system was to be able to express constraint propagation strategies more easily.
- Constraint Handling Rules (CHR) is a declarative high-level language extension especially designed for writing constraint solvers. With CHR, one can introduce *user-defined* constraints into a given host language, be it Prolog, Lisp, Java, or any other language.

Contribution of the Work

This work aims at covering all aspects of rule-based constraint programming. Going from theory to practice, we will present some analysis and debugging methods for rule-based constraint solvers. Then, we will present a new research area, where users for whom writing constraint solvers remains a hard task have the possibility to do it automatically. The paper then describes an implementation of a Java Constraint Kit consisting of a rule-based language for writing constraint solvers and a generic search engine to solve combinatorial problems. Then, we show how to extend a specific rule-based constraint language to become a general purpose language without losing the extra features supporting efficient constraint solving, e.g. committed choice and matching. Finally, we introduce two applications that benefit from using a rule-based constraint language for writing constraint solvers.

Since any language has its own characteristics, we only rely in this work on the high level language Constraint Handling Rules (CHR). One nice feature of CHR is to have constraints as a first class concept which is neither the case in CLAIRE nor in ELAN. With CHR, any first-order constraint theory and consistency algorithm can be implemented at a high-level of abstraction. Constraint solvers written in CHR (also called CHR programs) consist of guarded rules with multiple heads that replace constraints by simpler ones until they are solved. CHR defines both simplification of and propagation over user-defined constraints. Simplification rules replace constraints by simpler constraints while preserving logical equivalence (e.g., $\texttt{X>Y} \wedge \texttt{Y>X} \Leftrightarrow \texttt{false}$). Propagation rules add new constraints, which are logically redundant but may cause further simplification (e.g. $\texttt{X>Y} \wedge \texttt{Y>Z} \Rightarrow \texttt{X>Z}$). Repeated application of rules incrementally solves constraints. For example, with the two rules above we can transform $\texttt{A>B} \wedge \texttt{B>C} \wedge \texttt{C>A}$ to $\texttt{A>B} \wedge \texttt{B>C} \wedge \texttt{C>A} \wedge \texttt{A>C}$ and to $\texttt{false}$.

Analysis of Rule-based Constraint Solvers (Chapter 3)

Program analysis, both static and dynamic, is the central issue of programming environments. The declarativity of rule-based constraint languages makes it easier to analyze such solvers and to debug them. Firstly, we introduce some static analysis methods, i.e. analysis of *program source code*. We present important properties for constraint solvers, namely confluence and operational equivalence. Confluence means that the result of a computation is independent from the order in which rules are applied to the constraints. Operational equivalence of two programs means that for each goal, the final state in one program is the same as the final state in the other program. For confluence and operational equivalence of terminating CHR programs, we present decidable, sufficient and necessary syntactic conditions. To analyze constraint solvers dynamically, we present a tool, called VisualCHR, to support the analysis of *executions* as produced by CHR. The primary purpose of VisualCHR was to support developers of constraint solvers during the debugging stages. But VisualCHR also supports the comparison of different constraint solvers, or simply the illustration of the derivation process for teaching or learning purposes.

Automatic Generation of Constraint Solvers (Chapter 4)

Writing constraint solvers remains a hard task even in a rule-based formalism since the programmer has to determine the propagation algorithms. In this work, we propose a method to generate automatically the propagation and simplification process of constraints in form of rules. The rules will be implemented in CHR. The user has only to define the constraints extensionally and to determine the admissible syntactic form of the rules.

Consider the following example, where the user wants to generate a constraint solver for the boolean conjunction `and(X,Y,Z)`, where `X` and `Y` are the input arguments and `Z` is the output argument. This ternary relation can be defined extensionally by the triples {`(0,0,0)`, `(0,1,0)`, `(1,0,0)`, `(1,1,1)`}, where `1` stands for truth and `0` for falsity.

The propagation of the boolean conjunction can be specified by the following rules:

```
and(0,Y,Z) ⇔ Z=0.
and(X,0,Z) ⇔ Z=0.
and(1,Y,Z) ⇔ Y=Z.
and(X,1,Z) ⇔ X=Z.
and(X,X,Z) ⇔ X=Z.
and(X,Y,1) ⇔ X=1 ∧ Y=1.
```

For example, the first rule says that the constraint `and(X,Y,Z)`, when it is known that the first input argument `X` is equal to `0`, can be replaced by the constraint that the output `Z` must be equal to `0`. Hence, the goal `and(0,Y,Z)` will result in `Z=0`.

These rules are the well-known rules that can be found in several papers describing the propagation of boolean constraints, e.g. in form of demons [39], conditionals [99], CHR rules [45] or proof systems [36, 21]. Our aim is to provide a method to generate such rules automatically provided the user specifies the right hand side of the rules to be a conjunction of equality constraints.

Extension (Chapter 5)

The operational semantics of CHR differs from SLD resolution in various ways. Most of these differences are extensions, but there are also two incompatible differences:

- While SLD resolution performs full unification between goals and rule heads, CHR performs a one-sided unification ("matching"). That is, CHR only allows the variables of a rule to be instantiated in order to match the rule head with some part of the goal.
- While SLD resolution tries to apply all appropriate rules to a goal (usually implemented by backtracking), CHR applies only one, i.e., it follows the committed-choice approach.

These incompatibilities make CHR difficult to use as a general-purpose logic query language, especially for search-oriented problems.

We show, however, that only a small and simple extension to CHR is needed in order to reach an expressive power that subsumes the expressive power of Horn clause programs with SLD resolution: We allow disjunctions on the right hand sides of CHR rules. We call the extended language "$\mathrm{CHR}^\vee$" (to pronounce "CHR-or").

So while currently CHR is being used as a special-purpose language for constraint solvers and CHR programs are typically supplements to Prolog programs, $\mathrm{CHR}^\vee$ allows to write the entire application in a uniform language. But it is not only possible to merge constraint solving and top-down query evaluation: $\mathrm{CHR}^\vee$ allows also to write logic programs for bottom-up evaluation as it is frequently used in disjunctive deductive databases. Together with disjunction, it is even possible to implement disjunctive logic databases and to evaluate them in a bottom-up manner in the style of Satchmo [69] (formalized as PUHR tableaux [27]) and CPUHR tableaux [13]. Furthermore, abduction and integrity constraints can be expressed in $\mathrm{CHR}^\vee$ in a straightforward way. The contribution of $\mathrm{CHR}^\vee$ is, thus, a platform to experiment with several logic programming paradigms in a common implemented setting, rather than a new logic programming language.

Implementation (Chapter 6)

The constraint programming technology has matured to the point where it is possible to isolate some essential features and offer them as libraries or embedded cleanly in general purpose host programming languages. At the moment, most constraint systems are either extension of a programming language (often Prolog), e.g. Eclipse, or libraries which are used together with conventional programming language (often C or C++), e.g. ILOG Solver. Due to the growing popularity of Java and the possibilities of the Internet, there is a big interest to provide constraint handling in Java to implement application servers, e.g. for planning or scheduling systems.

In this work, we describe the design of a Java Constraint Kit, called JACK, consisting of a high-level language for writing constraint solvers (JCHR) and a generic search engine (JASE) to solve combinatorial problems. JCHR is an implementation of Constraint Handling Rules in Java. Because of the incomplete constraint propagation methods used for scheduling problems, the application programmer often has to explicitly use a labeling phase in which a backtracking search blindly tries different values for the variables. In JACK, search can be performed using JASE.

Applications (Chapter 7)

CHR is used by more than 40 projects worldwide and has been used to solve a wide range of applications. In this work, we present two applications, one solves a university timetabling problem, the other solves a room assignment problem. In both applications, we have to distinguish two kinds of constraints. *Hard constraints* are conditions that must be satisfied, *soft constraints* may be violated, but should be satisfied as far as possible. The classical approach to deal with these requirements is based on a variant of branch & bound search. In this work, we present new approaches dealing with soft constraints. In the first approach, we associate values with an estimate of how selecting a value to influence solution quality, i.e. which value is known (or expected) to violate soft constraints, or the other way round, which value is known (or expected) to satisfy soft constraints. In the second approach, the cost function used in branch & bound search and measuring the quality of a solution is computed during the constraint solving process.

Chapter 2

Constraint Programming

2.1 Constraint (Logic) Programming

Constraint programming is based on the idea that many interesting and difficult problems can be expressed declaratively in terms of variables and constraints. The variables range over a (finite) set of values and typically denote alternative decisions to be taken. The constraints are expressed as relations over subsets of variables and restrict admissible value combinations for the variables. Constraints can be given explicitly, by listing all possible tuples, or implicitly, by describing a relation in some (say mathematical) form. A solution is an assignment of variables to values which satisfies all constraints. Constraint programming can be expressed over many different domains like linear terms over rational numbers, Boolean algebra, finite/infinite sets or intervals over floating point numbers. Very interesting development is possible for most of these domains or more general domain independent constraint solvers.

Constraint logic programming (CLP) is the most developed of the constraint programming paradigms [55, 98, 50, 56]. In the last 15 years, CLP has evolved from a basic research idea to a powerful programming paradigm. CLP combines the declarativity of logic programming with the efficiency of constraint solving.

Constraint solving is the mechanism which controls the interaction of the constraints. Each constraint can deduce necessary conditions on the variable domains of its variables. The methods used for this constraint reasoning depend on the constraints, in the finite domain case they range from general but rather syntactic inference rules to complex combinations of algorithms used in the global constraints. Whenever a constraint updates a variable, the constraint propagation will wake all relevant constraints to detect further consequences.

In the beginning, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language, termed the "black-box" approach. While efficient, this approach makes it hard to modify a solver or build a solver over a new domain, let alone reason about and analyze it. As the behavior of the solver can neither be inspected by the user nor explained by the computer, debugging of constraint-based programs is hard. Also, one lesson learned from practical applications is that constraints are often heterogeneous and application specific. Several proposals have been made to allow more flexibility and customization of constraint solvers, often termed "glass-box" approaches [36, 98].

The most far-reaching proposal is the "no-box" approach: Constraint Handling Rules (CHR) [46].

2.2 Constraint Handling Rules

Constraint Handling Rules (CHR) [46] is a powerful special-purpose declarative programming language for writing constraint solvers either from scratch or by modifying existing solvers. CHR is essentially a committed-choice language consisting of multi-headed guarded rules that transform constraints into simpler ones until they are solved.

We now review the syntax and the declarative and operational semantics of CHR. For a more complete overview of CHR see [45].

2.2.1 Syntax

First-order terms, predicates, and atoms are defined in the usual way. We use two disjoint sorts of predicate symbols for two different classes of constraints: *built-in constraint symbols* and *user-defined constraint symbols (CHR symbols)*. We call an atomic formula with a constraint symbol an *atomic constraint* or simply a *constraint*. Built-in constraints are those handled by a predefined constraint solver that already exists. User-defined constraints are those defined by a CHR program.

A *CHR program* is a finite set of rules. There are three kinds of rules.
A *simplification rule* is of the form

$$Rulename \ @ \ H \Leftrightarrow C \mid B$$

a *propagation rule* is of the form

$$Rulename \ @ \ H \Rightarrow C \mid B$$

a *simpagation rule* is of the form

$$Rulename \ @ \ H_1 \backslash H_2 \Leftrightarrow C \mid B,$$

where *Rulename* is a unique identifier of a rule (names of rules are optional), the *head* H (or $H_1 \backslash H_2$) is a non-empty conjunction of user-defined constraints, the *guard* C is a conjunction of built-in constraints and the *body* B is a goal. A *goal* is a conjunction of built-in and user-defined constraints. A guard "*true*" is usually omitted together with the the commit symbol "$\mid$". A CHR symbol is *defined* in a CHR program if it occurs in the head of a rule in the program.

Since a simpagation rule is an abbreviation for the simplification rule

$$Rulename \ @ \ H_1 \wedge H_2 \Leftrightarrow C \mid H_1 \wedge B$$

there is no need to discuss them further in this section. However, we rely on simpagation rules later when we describe implementations and applications of CHR.

2.2.2 Declarative Semantics

The logical meaning of a simplification rule is a logical equivalence provided the guard holds $\forall \bar{x}\ (C \rightarrow (H \leftrightarrow \exists \bar{y}\ B))$[1]; the logical meaning of a propagation rule is an implication provided the guard holds $\forall \bar{x}\ (C \rightarrow (H \rightarrow \exists \bar{y}\ B))$, where $\bar{x}$ is the list of variables occurring in H and C and $\bar{y}$ are the variables occurring in B only. The logical meaning of a CHR program P is the conjunction of the logical meanings of its rules, $\mathcal{P}$, and a constraint theory CT that defines the built-in constraints. We require CT to define the predicate $=$ as syntactic equality.

2.2.3 Operational Semantics

The operational semantics of CHR can be described as a state transition system for *states* of the form $G_{\mathcal{V}}$, where G (the *goal*) is a conjunction of user-defined and built-in constraints and $\mathcal{V}$ a sequence of variables. The notation G_{built} and G_{user} denotes the built-in constraints and user-defined constraints, respectively, in a goal G.

We require that states are normalized so that they can be compared syntactically in a meaningful way. Basically, we require that the built-in constraints are in a (unique) normal form, where all syntactic equalities are made explicit and are propagated to the user-defined constraints. Furthermore, we require that the normalization projects out strictly local variables, i.e. variables appearing in the built-in constraints only.

Definition 2.1 [1, 7]
Let $\mathcal{N}$ be a function $\mathcal{N} : \mathcal{S} \rightarrow \mathcal{S}$, where $\mathcal{S}$ is the set of all states and let $G_{\mathcal{V}} \in \mathcal{S}$ and $\mathcal{N}(G_{\mathcal{V}}) = G'_{\mathcal{V}}$. Assume that there is a fixed order on variables appearing in a state such that the variables of $\mathcal{V}$ precede all other variables. Then $\mathcal{N}$ is a *normalization function*, if it fulfills the following conditions:

- *Equality propagation:* $G'_{\mathcal{V}}$ is obtained from $G_{\mathcal{V}}$ by replacing each variable x that is uniquely determined in G_{built}, i.e. for which $CT \models \forall\,(G_{built} \rightarrow x{=}t)$ holds, by the corresponding term t, except if t is a variable that comes after x in the variable order.

- *Projection:* The following must hold:

  $$CT \models \forall\ ((\exists \bar{x} G_{built}) \leftrightarrow G'_{built}),$$

  where $\bar{x}$ are the variables that appear in G_{built} but not in G'_{user}.

- *Uniqueness:* If

  $$\begin{array}{rcl} \mathcal{N}(G1_{\mathcal{V}}) & = & G1'_{\mathcal{V}} \text{ and} \\ \mathcal{N}(G2_{\mathcal{V}}) & = & G2'_{\mathcal{V}} \text{ and} \\ CT \models (\exists \bar{x} G1_{built}) & \leftrightarrow & (\exists \bar{y} G2_{built}), \end{array}$$

  holds, where $\bar{x}$ and $\bar{y}$, respectively, are the strictly local variables of the two states, then:

  $$G1'_{built} = G2'_{built}.$$

[1] $\forall F$ denotes the universal closure of a formula F

□

Given a CHR program P we define the transition relation $\mapsto_P$ by introducing two kinds of computation steps (Figure 2.1).

In Figure 2.1, an equation $c(t_1, \ldots, t_n){=}d(s_1, \ldots, s_n)$ of two constraints stands for $t_1{=}s_1 \wedge \ldots \wedge t_n{=}s_n$ if c and d are the same predicate symbols and for *false* otherwise. An equation $(p_1 \wedge \ldots \wedge p_n){=}(q_1 \wedge \ldots \wedge q_m)$ stands for $p_1{=}q_1 \wedge \ldots \wedge p_n{=}q_n$ if $n = m$ and for *false* otherwise. Conjuncts can be permuted since conjunction is associative and commutative.

Simplify

If	$(H \Leftrightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\, (G_{built} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_P \mathcal{N}((H{=}H' \wedge B \wedge C \wedge G))$

Propagate

If	$(H \Rightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\, (G_{built} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_P \mathcal{N}((H{=}H' \wedge B \wedge C \wedge H' \wedge G))$

Figure 2.1: Computation Steps of CHR

To **Simplify** user-defined constraints H' means to remove them from the state $H' \wedge G$ and to add the body B of a fresh variant of a simplification rule $(H \Leftrightarrow C \mid B)$ and the equation $H{=}H'$ and the guard C to the resulting state G, provided H' matches the head H and the guard C is implied by the built-in constraints appearing in G. In this case we say that the rule R is *applicable to* H'. A "variant" of a formula is obtained by renaming its variables. A "fresh" variant contains only new variables. "Matching" means that H' is an instance of H, i.e. it is only allowed to instantiate (bind) variables of H but not variables of H'. In the logical notation this is achieved by existentially quantifying only over the fresh variables $\bar{x}$ of the rule to be applied in the condition.

The **Propagate** transition is like the **Simplify** transition, except that it keeps the constraints H' in the state. Trivial nontermination caused by applying the same propagation rule again and again is avoided by applying a propagation rule at most once to the same constraints. A more complex operational semantics that addresses this issue can be found in [1].

$\mapsto_P^+$ denotes the transitive closure, $\mapsto_P^*$ denotes the reflexive and transitive closure of $\mapsto_P$.

An *initial state* for a goal G is the state $\mathcal{N}(G_\mathcal{V})$ where $\mathcal{V}$ is a sequence of all variables appearing in G. A *final state* is either of the form *false* (such a state is called *failed*) or of the form G with no computation step possible anymore and G not *false*. Final states containing user-defined constraints are called *blocking*. Final states of the form C, where C is a satisfiable built-in constraint, are called *successful*.

A *computation* of a goal G in a program P is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto_P S_{i+1}$ beginning with the initial state for G and ending in a final state or diverging. Where it is clear from the context, we will drop the reference to the program P.

Example 2.2 Let $\leq$ and $<$ be built-in constraint symbols. We define a user-defined constraint symbol `max`, where `max(X,Y,Z)` means that `Z` is the maximum of `X` and `Y`:

$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y} \mid \texttt{Z}=\texttt{Y}.$
$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}=\texttt{X}.$
$\texttt{max(X,Y,Z)} \Rightarrow \texttt{X}\leq\texttt{Z} \wedge \texttt{Y}\leq\texttt{Z}.$

The first rule states that `max(X,Y,Z)` can be simplified into `Z=Y` in any goal where it holds that $\texttt{X}\leq\texttt{Y}$. Analogously for the second rule. The third rule propagates constraints. It states that `max(X,Y,Z)` unconditionally implies $\texttt{X}\leq\texttt{Z} \wedge \texttt{Y}\leq\texttt{Z}$. Operationally, we add these logical consequences as redundant constraints, the `max` constraint is kept.

To the goal `max(1,2,M)` the first rule is applicable:
$\texttt{max}(1,2,\texttt{M}) \quad \mapsto \quad \texttt{M}=2.$

To the goal $\texttt{max(A,B,M)} \wedge \texttt{A}<\texttt{B}$ the first rule is applicable:
$\texttt{max(A,B,M)} \wedge \texttt{A}<\texttt{B} \quad \mapsto \quad \texttt{M}=\texttt{B} \wedge \texttt{A}<\texttt{B}.$

To the goal `max(A,A,M)` both simplification rules are applicable, and in both cases:
$\texttt{max}(\texttt{A},\texttt{A},\texttt{M}) \quad \mapsto \quad \texttt{M}=\texttt{A}.$

Redundancy from the propagation rule is useful, as the goal `max(A,3,3)` shows: To this goal only the propagation rule is applicable, but then the first rule:
$\texttt{max}(\texttt{A},3,3) \quad \mapsto \quad \texttt{max(A,3,3)} \wedge \texttt{A}\leq 3 \quad \mapsto \quad \mathcal{N}(3=3 \wedge \texttt{A} \leq 3) \; = \texttt{A}\leq 3.$

Note, that the constraint $3=3$ is simplified by the normalization function $\mathcal{N}$.
□

2.2.4 Soundness and Completeness

We now relate the operational and declarative semantics of CHR. These results are based on work of Jaffar and Lassez [55], Maher [68], and van Hentenryck [98]. The proofs for the following theorems can be found in [2].

Definition 2.3 Let A be a state which appears in a computation of G. The *logical meaning* of a state A is the formula

$$\exists \bar{y} A,$$

where $\bar{y}$ are the (local) variables appearing in A and not in the rest of G. The logical meaning of a final state is called *answer constraint*. □

The following results are based on the fact that the computation steps of CHR preserve the logical meaning of states. Lemma 2.4 says that all sates in a computation are logically equivalent.

Lemma 2.4 Let P be a CHR program and G be a goal. Then for all computable constraints C_1 and C_2 of G the following holds:

$$\mathcal{P}, CT \models \forall (C_1 \leftrightarrow C_2).$$

□

Theorem 2.5 (Soundness) Let P be a CHR program and G be a goal. If G has a computation with answer constraint C then

$$\mathcal{P}, CT \models \forall\ (C \leftrightarrow G).$$

$\square$

Theorem 2.6 (Completeness) Let P be a CHR program and G be a goal with at least one finite computation. If $\mathcal{P}, CT \models \forall\ (C \leftrightarrow G)$, then G has a computation with answer constraint C' such that

$$\mathcal{P}, CT \models \forall\ (C \leftrightarrow C').$$

$\square$

Theorem2.6 is stronger than the completeness result for CLP languages as presented in [68]. We can reduce the disjunction in the strong completeness theorem presented there to a single disjunct in our theorem. This is possible, since our declarative semantics is stronger and consequently, according to Lemma 2.4, all computable constraints of a given goal are equivalent

Chapter 3

Analysis of Rule-based Constraint Solvers

Program analysis, both static and dynamic, is the central issue of programming environments. Static analysis consists of analyzing the program source code. The role of dynamic analysis is to understand program execution. In this chapter, we present both static and dynamic analysis methods for CHR.

Previous work [6, 1, 7] have shown that static analysis techniques are available for an important property of any constraint solver, namely confluence: The result of a computation should be independent from the order in which constraints arrive and in which rules are applied to the constraints. For confluence of terminating CHR programs we were able to give a decidable, sufficient and necessary condition [1]. To make a non-confluent CHR program confluent by adding new rules, we also proposed a completion method [4]. In Section 3.1, we summarize previous confluence results. Detailed confluence results for simplification rules only are published in [6, 7]. These results have been simplified and extended to all kinds of rules in [1, 2].

A fundamental question in programming language semantics is when two programs should be considered equivalent. In Section 3.2, we introduce a notion of operational equivalence for CHR programs and user-defined constraints. We give a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent CHR programs [5].

To analyze CHR executions, we present a visualization tool, called VisualCHR, which supports the development of rule-based constraint solvers. VisualCHR can be used to debug and to improve the efficency of constraint solvers. It can also be used to understand the details of constraint propagation methods and the interaction of different constraints implemented by means of CHR. Thus, it is suitable for users at different levels of expertise.

3.1 Confluence

The confluence property of a program guarantees that any computation starting from an arbitrary initial state, i.e. any possible order of rule applications, results in the same final state. In the following, we just give an overview on confluence results for CHR programs, for details see [7, 1].

Definition 3.1 A CHR program is called *confluent* if for all states S, S_1, S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then S_1 and S_2 are joinable. Two states S_1 and S_2 are called *joinable* if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1, T_2 are variants of each other. □

To analyze confluence of a given CHR program we cannot check joinability starting from any given ancestor state S, because in general there are infinitely many such states. However one can restrict the joinability test to a finite number of "minimal" states based on the following observations: First, adding constraints to a state cannot inhibit the application of a rule as long as the built-in constraints remain consistent (monotonicity property, cf. Lemma 3.17 in Section 3.2.2). Hence we can restrict ourselves to ancestor states that consist of the head and guards of two rules. Second, joinability can only be destroyed if one rule inhibits the application of another rule. Only the removal of constraints can affect the applicability of another rule, in case the removed constraint is needed by the other rule. Hence at least one rule must be a simplification rule and the two rules must *overlap*, i.e. have at least one head atom in common in the ancestor state. This is achieved by equating head atoms in the state.

Definition 3.2 Given a simplification rule R_1 and an arbitrary (not necessarily different) rule R_2, whose variables have been renamed apart. Let $H_i \wedge A_i$ be the head and C_i be the guard of rule R_i $(i = 1, 2)$. Then a *critical ancestor state of R_1 and R_2* is

$$(H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2) \wedge C_1 \wedge C_2)_{\mathcal{V}},$$

provided A_1 and A_2 are non-empty conjunctions and $CT \models \exists((A_1{=}A_2) \wedge C_1 \wedge C_2)$.

□

The application of R_1 and R_2, respectively, to a critical ancestor state of R_1 and R_2 leads to two states that form the so-called *critical pair*.

Definition 3.3 Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2 then the tuple (S_1, S_2) is a *critical pair* of R_1 and R_2. A critical pair (S_1, S_2) is *joinable*, if S_1 and S_2 are joinable.

□

Definition 3.4 A CHR program is called *terminating*, if there are no infinite computations. □

In general, testing the termination of a CHR program is undecidable. However, for most existing CHR programs it is straightforward to prove termination using simple well-founded orderings [47].

The following theorem from [6, 1, 7] gives a decidable, sufficient and necessary condition for confluence of a terminating CHR program:

Theorem 3.5 A terminating CHR program is confluent iff all its critical pairs are joinable.

□

Example 3.6 Consider the program for `max` of Example 2.2. The following critical pair stems from the critical ancestor state[1] $(\texttt{max(X,Y,Z)} \wedge \texttt{X} \leq \texttt{Y})_{[\texttt{X,Y,Z}]}$ of the first rule and the third one:

$$(S_1, S_2) := (\texttt{Z=Y} \wedge \texttt{X}\leq\texttt{Y} \quad , \quad \texttt{max(X,Y,Z)} \wedge \texttt{X}\leq\texttt{Y} \wedge \texttt{X}\leq\texttt{Z} \wedge \texttt{Y}\leq\texttt{Z})$$

(S_1, S_2) is joinable since S_1 is a final state and the application of the first rule to S_2 results in S_1.

□

3.2 Operational Equivalence

A fundamental and hard question in programming language semantics is when two programs should be considered equivalent. For example correctness of program transformation can be studied only with respect to a notion of equivalence. Also, if modules or libraries with similar functionality are used together, one may be interested in finding out if program parts in different modules or libraries are equivalent. In the context of CHR, this case arises frequently when constraint solvers written in CHR are combined. Typically, a constraint is only partially defined in a constraint solver. We want to make sure that the operational semantics of the common constraints of two programs do not differ, and we are interested in finding out if they are equivalent.

For example, we would like to know if the following two CHR rules defining the user-defined constraint `max`

```
max(X,Y,Z) ⇔ X<Y | Z=Y.
max(X,Y,Z) ⇔ X≥Y | Z=X.
```

are operationally equivalent with these two rules

```
max(X,Y,Z) ⇔ X≤Y | Z=Y.
max(X,Y,Z) ⇔ X>Y | Z=X.
```

or if the union of the rules results in a better constraint solver for `max`.

The literature on equivalence of programs in logic-based languages is sparse. In most papers that touch the subject, a suitable notion of program equivalence serves as a correctness criterion for transformations between programs, e.g. in partial evaluation and deduction. Our concern is the problem of program equivalence in its generality, where the programs to be compared are independent from each other.

[67] provides a systematic comparison of the relative strengths of various formulations of equivalence of logic programs. These formulations arise naturally from several formal semantics of logic programs. Maher does not study how to test for equivalence. The results may be extensible to constraint logic programs, but committed-choice languages like CHR have different semantics that induce different notions of equivalence. In particular, in CHR the distinction between successful, failed or deadlocked goals is secondary, but the distinction between a goal and its instances is vital. For similar reasons, [51] among other

[1] For readability, variables from different rules have been identified to have an overlap.

things extends Maher's work by considering relationships between equivalences derived from semantics that are based e.g. on computed answer substitutions. Gabbrielli et. al. are not concerned with tests for equivalence, either.

Like [51] we are concerned with equivalences of the observable behavior of programs. Observables are then a suitable abstraction of execution traces. In case of equivalence based on operational semantics expressed by a transition system, it is common to define as observables the results of finite computations, where one abstracts away local variables, see e.g. [41].

The following definition states that two CHR programs are operationally equivalent if for each goal, the final state in one program is the same as the final state in the other program.

Definition 3.7 Let P_1 and P_2 be CHR programs. A state S is P_1, P_2*-joinable*, iff there are two computations $S \mapsto^*_{P_1} S_1$ and $S \mapsto^*_{P_2} S_2$, where S_1 and S_2 are final states, and S_1 and S_2 are variants of each other.

Let P_1 and P_2 be CHR programs. P_1 and P_2 are *operationally equivalent* if all states are P_1, P_2-joinable. □

It is tempting to think that a suitable modification of the concept of confluence can be used to express equivalence of programs. In Section 3.2.1, we show that a straightforward application of our confluence test is too weak to capture the operational equivalence of CHR programs.

In practice, one is often interested in comparing implementations of constraints instead of whole programs. Hence we investigate a notion of operational equivalence for user-defined constraints that are defined in different programs. We give a sufficient syntactic condition for constraints defined in terminating and confluent CHR programs (Section 3.2.2). For a subclass of programs which have only one user-defined constraint in common, we are able to give a sufficient and necessary syntactic condition.

Based on these results, we are finally able to give a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent CHR programs (Section 3.2.3).

3.2.1 Compatibility of Programs

We can use our confluence test to ensure that the different, confluent programs are "compatible": The union of the programs is confluent.

Definition 3.8 Let P_1 and P_2 be two confluent and terminating CHR programs and let the union of the two programs, $P_1 \cup P_2$, be terminating. P_1 and P_2 are *compatible* if $P_1 \cup P_2$ is confluent. □

Testing the compatibility of P_1 and P_2 means to test the joinability of the critical pairs of $P_1 \cup P_2$, i.e. the critical pairs of P_1 united with the critical pairs of P_2 united with critical pairs coming from one rule in P_1 and one rule in P_2, and to test the termination of $P_1 \cup P_2$. Note that critical pairs from rules of different programs can only exist, if the heads of the rules have at least one constraint in common.

If the confluence test fails, we can locate the rules responsible for the problem. If the test succeeds, we can just take the union of the rules in the two programs. This means that a common CHR symbol can even be partially defined in the programs which are combined.

Example 3.9 P_1 contains the following CHR rules defining `max`:

```
max(X,Y,Z) ⇔ X<Y | Z=Y.
max(X,Y,Z) ⇔ X≥Y | Z=X.
```

whereas P_2 has the following definition of `max`:

```
max(X,Y,Z) ⇔ X≤Y | Z=Y.
max(X,Y,Z) ⇔ X>Y | Z=X.
```

We want to know whether the definitions of `max` are compatible. There are three critical ancestor states coming from one rule in P_1 and one rule in P_2:

- `max(X,Y,Z)` $\wedge$ `X<Y` $\wedge$ `X≤Y` stems from the first rule of P_1 and the first rule of P_2.
- `max(X,Y,Z)` $\wedge$ `X≥Y` $\wedge$ `X≤Y` stems from the second rule of P_1 and the first rule of P_2.
- `max(X,Y,Z)` $\wedge$ `X≥Y` $\wedge$ `X>Y` stems from the second rule of P_1 and the second rule of P_2.

Since the critical pairs coming from the critical ancestor states described above are joinable, the two definitions of `max` are compatible. Hence we can just take the union of the rules and define `max` by all four rules.

Note that the compatibility test does not ensure that the constraints are operationally equivalent. In P_1 the goal `max(X,Y,Z)` $\wedge$ `X≥Y` has the following computation:

$$\texttt{max(X,Y,Z)} \wedge \texttt{X}\geq\texttt{Y} \quad \mapsto_{P_1} \quad \texttt{Z=X} \wedge \texttt{X}\geq\texttt{Y}$$

In P_2 the initial state `max(X,Y,Z)` $\wedge$ `X≥Y` is also final state, i.e. no computation step is possible. On the other hand, in P_2 the goal `max(X,Y,Z)` $\wedge$ `X≤Y` has a non-trivial computation, while the goal is a final state in P_1.

The constraint `max` is "operationally stronger" in $P_1 \cup P_2$ than in each program alone, in the sense that more computation steps are possible.

□

3.2.2 Equivalence of Constraints

We now introduce a test to ensure that the definitions of the same CHR symbol in different programs are not only compatible, but indeed are operationally equivalent. We first restrict our attention to states that consist of one CHR symbol (only) being common to both programs.

Definition 3.10 Let c be a CHR symbol. A *c-state* is a state where all user-defined constraints have the same CHR symbol c. □

Definition 3.11 Let c be a CHR symbol defined in two CHR programs P_1 and P_2. P_1 and P_2 are *operationally c-equivalent* if all c-states are P_1, P_2-joinable. □

We give now a sufficient syntactic condition for operational c-equivalence of terminating CHR programs. As with confluence, we will try to find a finite subset of states, such that the P_1, P_2-joinability of the subset implies P_1, P_2-joinability of all c-states. As we will see, the similarities with confluence will not go much beyond that, mainly because in operational c-equivalence two different programs are involved.

The following example illustrates, that, first of all, the critical pairs known from confluence (and compatibility) are not the right subset of states to ensure operational equivalence.

Example 3.12 Let P_1 be the following CHR program:

```
p(a) ⇔ s.
p(b) ⇔ r.
s∧r ⇔ true.
```

and let P_2 consist only of the first two rules.

It is not sufficient for operational equivalence to consider the critical pairs coming from the critical ancestor states `p(a)` and `p(b)`: In P_1 the conjunction `p(a)` $\wedge$ `p(b)` leads to `true`, but in P_2 the goal `s∧r` is a final state. □

The example indicates that we not only have to consider c-states, but also those states that can be reached from c-states. Because even if these states can be reached in different programs due to confluence and even if they are final states, there may be contexts (extensions of the states by more constraints) in which the computation can be continued, and it can be continued in different ways in the different programs. The idea is to avoid this by making sure that also the user-defined constraints that occur in these states are operationally equivalent.

For a given CHR symbol c one can safely approximate the set of all CHR symbols that appear in successor states to a c-state by looking at the bodies of rules with c in the head. Based on this idea we introduce the notion of dependency between CHR symbols.

Definition 3.13 A CHR symbol c *depends directly* on a CHR symbol c', if there is a rule in whose head c appears and in whose body c' appears. A CHR symbol c *depends* on a CHR symbol c', if c depends directly on c', or if c depends directly on a CHR symbol d and d depends on c'.

The *dependency set* of a CHR symbol c is the the set of all CHR symbols that c depends on. Let C_{P_1}, C_{P_2} be the dependency sets of c with respect to P_1 and P_2, respectively. Each CHR symbol from $(C_{P_1} \cap C_{P_2}) \cup \{c\}$ is called a *c-dependent CHR symbol.* □

Definition 3.14 Let P_1 and P_2 be CHR programs. The set of *c-critical states* is defined as follows:

$$\{H \wedge C \mid (H \odot C \mid B) \in P_1 \cup P_2, \text{ where } \odot \in \{\Leftrightarrow, \Rightarrow\} \text{ and } \\ H \text{ contains only } c\text{-dependent CHR symbols}\}$$

□

The set of c-critical states is formed by taking the head and guards of all rules in whose heads c-dependent CHR symbols appear.

In the following we will show that P_1, P_2-joinability of these minimal states is sufficient for P_1, P_2-joinability of arbitrary c-states. Before we can state and prove the theorem, we need several lemmata.

The first lemma states that normalization has no influence on applicability of rules. We therefore can assume in the following that states are normalized except where otherwise noted.

Lemma 3.15 Let S and S' be states.

$S \mapsto S'$ holds iff $\mathcal{N}(S) \mapsto S'$.

Proof: Can be found in [7]. □

The following lemma shows that a computation can be repeated in any context, i.e. with states in which constraints have been added.

Definition 3.16 The pair of constraints (G_1, G_2) is called *connected via* $\mathcal{V}$ iff all variables that appear both in G_1 and in G_2 also appear in $\mathcal{V}$. □

Lemma 3.17 [Monotonicity] If (G, H) is connected via $\mathcal{V}'$ and $G_{\mathcal{V}} \mapsto^* G'_{\mathcal{V}}$, and $\mathcal{V} \subseteq \mathcal{V}'$, then

$(G \wedge H)_{\mathcal{V}'} \mapsto^* \mathcal{N}((G' \wedge H)_{\mathcal{V}'})$.

Proof: Can be found in [7]. □

Next we show that a computation can be repeated in a state where variables have been instantiated according to some equations.

Definition 3.18 Let C be a conjunction of built-in constraints. Let H and H' be conjunctions of user-defined constraints with disjoint variables. $C[H{=}H']$ is obtained from C by replacing all variables x by the corresponding term t, where $CT \models H{=}H' \rightarrow (x{=}t)$ and x appears in H and t appears in H'. □

Lemma 3.19 Let P be a CHR program and let R be a rule from P with head H and guard C. Let H' be a conjunction of user-defined constraints. Let $(H \wedge H{=}H' \wedge C)_{\mathcal{V}}$ and $(H' \wedge C[H{=}H'])_{\mathcal{V}'}$ be initial states, where H and H' have disjoint variables. If $CT \models \exists \bar{x}(H = H' \wedge C)$, where $\bar{x}$ are the variables appearing in H, and $(H \wedge H = H' \wedge C)_{\mathcal{V}} \mapsto^*_P G_{\mathcal{V}}$, then $(H' \wedge C[H{=}H'])_{\mathcal{V}'} \mapsto^*_P G_{\mathcal{V}'}$.
Proof: The claim holds due to the equality propagation property of the normalization function $\mathcal{N}$ and according to Lemma 3.15. A detailed proof can be found in [5]. □

Next we show that a computation can be repeated in a state where redundant built-in constraints have been removed.

Lemma 3.20 Let C be a conjunction of built-in constraints. If $H \wedge C \wedge G \mapsto^* S$ and $CT \models \forall\, (G_{built} \rightarrow C)$ then $H \wedge G \mapsto^* S$.
Proof: This is a consequence of the following claim: If $H \wedge C \wedge G \mapsto S$ and $CT \models \forall\, (G_{built} \rightarrow C)$ then $H \wedge G \mapsto S$. This claim can be proven by analyzing each kind of computation step [5]. □

Finally, the last Lemma refers to joinability of c-critical states.

Definition 3.21 Let $C = \bigwedge_{i=1}^{n} C_i$ be a conjunction of constraints, π a permutation on $[1, \ldots, n]$, where $0 \leq m \leq n$, then $\bigwedge_{i=1}^{m} C_{\pi_i}$ is a *subconjunction* of C. □

Lemma 3.22 Let P_1 and P_2 be terminating CHR programs defining a CHR symbol c and let G be a goal. If all c-critical states are P_1, P_2-joinable and there is a rule in P_1 that is applicable to G_{user} then there is a rule in P_2 that is applicable to a subconjunction of G_{user}.

Proof:

We prove the claim by contradiction. We assume that there is a rule R_1 in P_1 that is applicable to G_{user} but there is no rule R_2 in P_2 that is applicable to a subconjunction of G_{user}. Let H_1 be the head of R_1 and let C_1 be its guard.

Since all c-critical states are P_1, P_2-joinable, $H_1 \wedge C_1$ is P_1, P_2-joinable, i.e. $H_1 \wedge C_1 \mapsto^*_{P_1} S$ and $H_1 \wedge C_1 \mapsto^*_{P_2} S$, where S is a final state. Since the program is terminating, S is different from $H_1 \wedge C_1$. Then there is a rule R_2 in P_2 with head H_2 and guard C_2 that is applicable to a subconjunction H_S of H_1, i.e. $CT \models C_1 \rightarrow \exists \bar{x}(H_2 = H_S \wedge C_2)$. Since R_1 is applicable to G_{user}, $CT \models G_{built} \rightarrow \exists \bar{y}(H_1 = G_{user} \wedge C_1)$. Then the following holds $CT \models G_{built} \rightarrow \exists \bar{x}(H_2 = H'_S \wedge C_2)$, where H'_S is a subconjunction of G_{user}. This contradicts the assumption.

□

We are now ready to state and prove the main theorem that gives a sufficient condition for operational c-equivalence.

For the proof of Theorem 3.24 to go through, CHR programs have to satisfy a *range-restriction* condition: In every rule, every variable in the body appears also in the head, i.e. there are no local variables. Nevertheless, our theorem holds for general CHR programs using the same proof technique, but the proof would be longish and cluttered with technicalities taking into account local variables. The proof can be found in [5].

The proof is by induction on the number of so-called macro-steps in a computation. These are conveniently chosen non-empty, finite sub-computations, as the following definition shows:

Definition 3.23 Let R be a CHR rule with head H and guard C and let $(H \wedge C)_{\mathcal{V}} \mapsto^+ B_{\mathcal{V}}$ be a computation, where $B_{\mathcal{V}}$ is a final state. Let $H' \wedge G$ be a goal and let R be applicable to H'. A *macro step* of a goal $H' \wedge G$ is a computation of the form $(H' \wedge G)_{\mathcal{V}'} \mapsto^+ \mathcal{N}((B \wedge H = H' \wedge G)_{\mathcal{V}'})$. □

Theorem 3.24 Let c be a CHR symbol defined in two confluent and terminating CHR programs P_1 and P_2. Then the following holds: P_1 and P_2 are operationally c-equivalent if all c-critical states are P_1, P_2-joinable.

Proof:

Using Lemma 3.22 we can show that the number of macro steps in a computation of a goal G in P_1 and P_2 are equal. Since P_1 and P_2 are terminating, the number of macro steps in these computations is finite.

In order to prove that P_1 and P_2 are operationally c-equivalent, we prove by induction over the number of macro steps that the final states of a goal G in P_1

and P_2, respectively, are equal. Since P_1 and P_2 are confluent, any computation for the goal G will lead to the same final state.

Base case: $n = 0$. G is a final state for P_1 and P_2, i.e. no rule is applicable.

Induction step: We assume that the induction hypothesis holds for $m < n$. We prove the assertion for n.

Let G be of the form $H' \wedge G'$ and there is a rule R in P_1 that is applicable to H', then according to Lemma 3.22 there is a rule in P_2 which is applicable to a subconjunction of H'.

Let R have head H and guard C. Then there is a computation of the form $(H \wedge C)_{\mathcal{V}} \mapsto^*_{P_1} B_{\mathcal{V}}$, where B is a final state. In the following we use the assumption that all c-critical states are P_1, P_2-joinable: There is also a computation $(H \wedge C)_{\mathcal{V}} \mapsto^*_{P_2} B$, where B is a final state.

Let G' be of the form $G'_{built} \wedge G'_{user}$, where G'_{built} and G'_{user} are conjunctions of built-in and user-defined constraints, respectively. Then the following holds:

- According to Lemma 3.17:
  $(H \wedge C \wedge H{=}H')_{\mathcal{V}_1} \mapsto^*_{P_1} \mathcal{N}((B \wedge H{=}H')_{\mathcal{V}_1})$
- According to Lemma 3.19:
  $(H' \wedge C[H{=}H'])_{\mathcal{V}_2} \mapsto^*_{P_1} \mathcal{N}((B \wedge H{=}H')_{\mathcal{V}_2})$
- According to Lemma 3.17:
  $(H' \wedge C[H{=}H'] \wedge G'_{built})_{\mathcal{V}_3} \mapsto^*_{P_1} \mathcal{N}((B \wedge H{=}H' \wedge G'_{built})_{\mathcal{V}_3})$
- Since R is applicable to H', the applicability condition $CT \models \forall\, (G'_{built} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$ holds. $CT \models \forall\, (\exists \bar{x}(H{=}H' \wedge C) \rightarrow C[H{=}H'])$ holds, hence $CT \models \forall\, (G'_{built} \rightarrow C[H{=}H'])$ holds. Therefore, according to Lemma 3.20, the built-in constraint $C[H{=}H']$ can be removed from the state and the same rules remain applicable:
  $(H' \wedge G'_{built})_{\mathcal{V}_3} \mapsto^*_{P_1} \mathcal{N}((B \wedge H{=}H' \wedge G'_{built})_{\mathcal{V}_2})$
- According to Lemma 3.17 we can add the constraints G'_{user}:
  $(H' \wedge G'_{built} \wedge G'_{user})_{\mathcal{V}_4} \mapsto^*_{P_1} \mathcal{N}((B \wedge H{=}H' \wedge G'_{built} \wedge G'_{user})_{\mathcal{V}_4})$
- Since $G' = G'_{built} \wedge G'_{user}$ the following holds:
  $(H' \wedge G')_{\mathcal{V}_4} \mapsto^*_{P_1} \mathcal{N}((B \wedge H{=}H' \wedge G')_{\mathcal{V}_4})$

By the same argumentation as above the following holds:
$(H' \wedge G')_{\mathcal{V}_4} \mapsto^*_{P_2} \mathcal{N}((B \wedge H{=}H' \wedge G')_{\mathcal{V}_4})$.

The number of macro steps in a computation for the goal $B \wedge H{=}H' \wedge G'$ is $n-1$. By the induction hypothesis and according to Lemma 3.15 the following holds:
$\mathcal{N}((B \wedge H{=}H' \wedge G')_{\mathcal{V}_4}) \mapsto^*_{P_1} S_F$ and $\mathcal{N}((B \wedge H{=}H' \wedge G')_{\mathcal{V}_4}) \mapsto^*_{P_2} S_F$.

The final states of G in P_1 and P_2, respectively, are equal, i.e. S_F.

□

We now give an example of two operationally equivalent user-defined constraints.

Example 3.25 The constraint `sum(List,Sum)` holds if `Sum` is the sum of elements of a given list `List`. The CHR symbol `sum` can be implemented in different ways.

Let P_1 be the following CHR program:

```
sum([],Sum) ⇔ Sum=0.
sum([X|Xs],Sum) ⇔  sum(Xs,Sum1) ∧ Sum = Sum1 + X.
```

Let P_2 be a CHR program that implements `sum` using an auxiliary CHR symbol `sum1`:

```
sum([],Sum) ⇔ Sum = 0.
sum([X|Xs],Sum) ⇔  sum1(X,Xs,Sum).
sum1(X,[],Sum) ⇔ Sum = X.
sum1(X,Xs,Sum) ⇔ sum(Xs,Sum1) ∧ Sum = Sum1 + X.
```

There are two `sum`-critical states coming from P_1 and P_2: `sum([],Sum)` and `sum([X|Xs],Sum)`. These `sum`-critical states are P_1, P_2-joinable:

For the `sum`-critical state `sum([],Sum)` the final state is `Sum = 0` in both P_1 and P_2.

A computation of the `sum`-critical state `sum([X|Xs],Sum)` in P_1 proceeds as follows:

$$\mathtt{sum([X|Xs],Sum)} \ \mapsto_{P_1} \ \mathtt{sum(Xs,Sum1)} \wedge \mathtt{Sum} = \mathtt{Sum1} + \mathtt{X}$$

A computation of the same initial state in P_2 results in the same final state:

$$\mathtt{sum([X|Xs],Sum)} \mapsto_{P_2} \mathtt{sum1(X,Xs,Sum)} \mapsto_{P_2} \mathtt{sum(Xs,Sum1)} \wedge \mathtt{Sum} = \mathtt{Sum1} + \mathtt{X}$$

Since all `sum`-critical states are P_1, P_2-joinable, P_1 and P_2 are operationally `sum`-equivalent. □

The next example shows why our joinability test for critical states is a sufficient, but not necessary condition for operational equivalence.

Example 3.26 Let P_1 be the following CHR program

```
p(X) ⇔  X>0 | q(X).
q(X) ⇔  X<0 | true.
```

and let P_2 be the following one

```
p(X) ⇔  X>0 | q(X).
q(X) ⇔  X<0 | false.
```

P_1 and P_2 are operationally `p`-equivalent, but the `p`-critical state `q(X)` ∧ `X<0` is not P_1, P_2-joinable.

□

The reason that we can only give a sufficient, but not necessary condition for operational c-equivalence in the general class of CHR programs is that the dependency relation between user-defined constraints only approximates the actual set of user-defined constraints that occur in states that can be reached from a c-state.

A sufficient and necessary condition: In practice, one is often interested to compare constraint solvers which have only one CHR symbol in common. In this case we can give a decidable, sufficient and necessary condition.

Theorem 3.27 Let c be the only CHR symbol defined in two confluent and terminating CHR programs P_1 and P_2. P_1 and P_2. Then the following holds: P_1 and P_2 are operationally c-equivalent iff all c-critical states are P_1, P_2-joinable.
Proof:

"$\Longrightarrow$" direction: Let P_1 and P_2 be operationally c-equivalent. We prove by contradiction that all c-critical states are P_1, P_2-joinable: Assume that $H \wedge C$ is a c-critical state that is not P_1, P_2-joinable, where H is the head of a rule from $P_1 \cup P_2$ and C its guard.

Since P_1 and P_2 have only c in common, the constraint symbol c is the only c-dependent CHR symbol, i.e. $(C_{P_1} \cap C_{P_2}) \cup \{c\} = \{c\}$. Therefore $H \wedge C$ is a c-state. This contradicts the prerequisite that P_1 and P_2 are operationally c-equivalent.

"$\Longleftarrow$" direction: This is a special case of Theorem 3.24.

□

Theorem 3.27 gives a decidable characterization of the c-equivalent subset of terminating and confluent CHR programs: P_1, P_2-joinability of a given c-critical state is decidable for a terminating CHR program and there are only finitely many c-critical states.

Example 3.28 The user-defined constraint `range(X,Min,Max)` holds if `X` is between `Min` and `Max`.

Let P_1 be a CHR program that implements `range` using the CHR symbol `max`:

```
max(X,Y,Z) ⇔ X<Y | Z=Y.
max(X,Y,Z) ⇔ X≥Y | Z=X.

range(X,Min,Max) ⇔ max(X,Min,X) ∧ max(X,Max,Max).
```

Let P_2 be a program defining `range` using the built-in constraint symbols $<$ and $\leq$:

```
range(X,Min,Max) ⇔ Max<Min | false.
range(X,Min,Max) ⇔ Min≤Max | Min≤X ∧ X≤Max.
```

P_1 and P_2 are not operationally `range`-equivalent, since the `range`-critical state `range(X,Min,Max)` coming from P_1 is not P_1, P_2-joinable: `range(X,Min,Max)` can be reduced to `max(X,Min,X) ∧ max(X,Max,Max)` in P_1. In P_2 the answer for the state `range(X,Min,Max)` is the state itself, because no rule is applicable.

P_1 is "operationally stronger" than P_2, since the computation step in P_1 does not require that the values of `Max` and `Min` are known. This can be exemplified by the goal `range(5,6,Max)`. The inconsistency of the goal can be detected in P_1. In P_2, `range(5,6,Max)` is a final state.

□

3.2.3 Equivalence of Programs

Based on the condition presented above for the operational equivalence of constraints we can also give a decidable, sufficient and necessary condition for operational equivalence of terminating and confluent programs.

However, it is not enough to consider the union of all c-critical states for all common CHR symbols c, as the following example illustrates.

Example 3.29 Let P_1 be

```
p ⇔ s.
s∧q ⇔ true.
```

and let P_2 be

```
p ⇔ s.
s∧q ⇔ false.
```

P_1 and P_2 have three common CHR symbols, `p`, `s` and `q`. `s` and `p` are the `p`-dependent constraint symbols. There are no `s`-dependent CHR symbols except `s` itself. Analogously for `q`.

`p` is the only `p`-critical state. It is P_1, P_2-joinable. There is no `s`-critical state, since `q` is not a `s`-dependent CHR symbol. Analogously for `q`.

Hence all `p`-, `s` and `q`-critical states are P_1, P_2-joinable, but the programs are not operationally equivalent. `s∧q` leads in P_1 to `true` and with P_2 to `false`. □

Still we can prove the operational equivalence of two programs by adapting the definition of c-critical states:

Definition 3.30 Let P_1 and P_2 be CHR programs. The *set of critical states of* P_1 *and* P_2 is defined as follows:

$$\{H \wedge C \mid (H \ \odot \ C \mid B) \in P_1 \cup P_2, \text{ where } \odot \in \{ \Leftrightarrow , \Rightarrow \}\}$$

□

Theorem 3.31 Let P_1 and P_2 be terminating and confluent programs. P_1 and P_2 are operationally equivalent iff all critical states of P_1 and P_2 are P_1, P_2-joinable.

Proof: Follows the proof of Theorem 3.27. □

3.2.4 Relationships

Operational equivalence of two confluent and terminating CHR programs implies their compatibility, since operational equivalence of P_1 and P_2 implies the confluence of $P_1 \cup P_2$. The converse does not hold, as the programs of Example 3.9 show.

Furthermore, operational equivalence of two CHR programs implies the operational c-equivalence of all common constraints, since the set of critical states is a superset of the union of all sets of the c-critical states. The converse does not hold, as the programs of Example 3.29 show.

3.3 Visualization

CLP presents in many cases advantages over imperative programming or other declarative paradigms. Nevertheless, CLP is not as widely used as it should by industrials. One of the factors that can presumably make the use of CLP more pervasive by industry is the availability of advanced programming environments which facilitate the development, debugging and exploitation of systems based on this paradigm.

In particular, the development of applications using early CLP systems has pointed out the need for studying CLP specific debugging and visualization techniques [16]. Current constraint visualization tools focus on representing and analyzing the search tree of a constraint program [90, 72, 91]. There is a lack of intuitive interactive tools for debugging the behavior of constraint solvers.

The contribution of this work is the development of a tool, called VisualCHR, to support the development of constraint solvers written in CHR. VisualCHR can be used to debug and to improve the efficency of constraint solvers. It can also be used to understand the details of constraint propagation methods and the interaction of different constraints implemented by means of CHR. Thus, it is suitable for users at different levels of expertise.

We will illustrate the visualization tool of CHR by the following example.

Example 3.32 We define a user-defined constraint for a (partial) order `leq` that can handle variable arguments.

```
Reflexivity  @ leq(X,X) ⇔  true.
Antisymmetry @ leq(X,Y) ∧ leq(Y,X) ⇔ X=Y.
Transitivity @ leq(X,Y) ∧ leq(Y,Z) ⇒ leq(X,Z).
```

The CHR program implements reflexivity, antisymmetry and transitivity in a straightforward way. The reflexivity rule states that `leq(X,X)` is logically true. The antisymmetry rule means that if we find `leq(X,Y)` as well as `leq(Y,X)` in the current store, we can replace them by the logically equivalent `X=Y`. The transitivity rule propagates constraints. It states that the conjunction of `leq(X,Y)` and `leq(Y,Z)` implies `leq(X,Z)`. Operationally, we add the logical consequence `leq(X,Z)` as a redundant constraint.

□

3.3.1 Representation of the Constraint Store

In constraint programming, the constraint store stores information about variables expressed by constraints and the constraint solver tries to simplify the store by constraint propagation and simplification. The constraint propagation and simplification in CHR is defined by rules.

The visualization of the constraint propagation depends on the representation of the store. A constraint store can be represented graphically by a box consisting of all its constraints. We call such representation *box view*. In Figure 3.1, the goal `leq(X,Y) ∧ leq(Z,X) ∧ leq(Y,Z)` is represented in a box view.

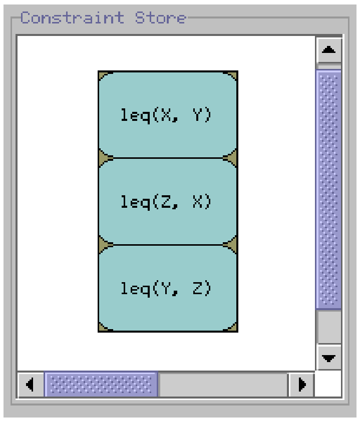


Figure 3.1: Box View of a Constraint Store

Additionally, a constraint store can be represented by a set of sub-boxes, where each sub-box consists of only one constraint. We call such representation *sub-box view*. In Figure 3.2, the goal `leq(X,Y)` $\wedge$ `leq(Z,X)` $\wedge$ `leq(Y,Z)` is represented in a sub-box view.

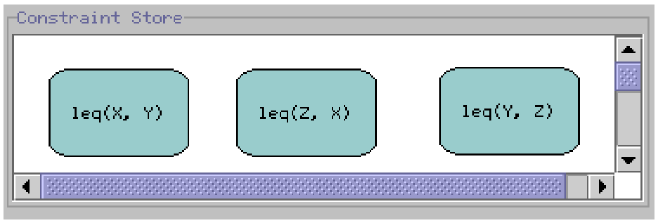


Figure 3.2: Sub-Box View of a Constraint Store

3.3.2 Creation and Expansion of Graphs

Using a box view, the constraint propagation will be visualized by a *linear sequence*. However, the visualization of constraints represented in a sub-box view leads to a graph.

Initially, in a sub-box view of a constraint store the graph consists of the nodes representing the goal, i.e. each node corresponds to a constraint. VisualCHR provides an operation "next", which uses the built-in inference engine to expand the graph by applying rules applicable to nodes in the graph. For the `leq` example, the transitivity rule is applicable on `leq(X,Y)` $\wedge$ `leq(Y,Z)` and its body results in a node `leq(X,Z)` of the graph (Figure 3.3). The constraints `leq(X,Y)` and `leq(Y,Z)` remain in the constraint store since the transitivity rule is a propagation rule. To distinguish between constraints which are removed by simplification rules and constraints remaining in the constraint store, we use different colors, i.e. orange for removed constraints and blue for constraints which remain in the constraint store. Since in general more than one constraint may cause a rule to fire and more than one constraint can be added to the constraint store in one rule application, we decided to represent the rule names as a node of a graph. These nodes are clickable, i.e. by clicking these nodes the whole rule as it appears in the constraint solver is displayed (Figure 3.4).

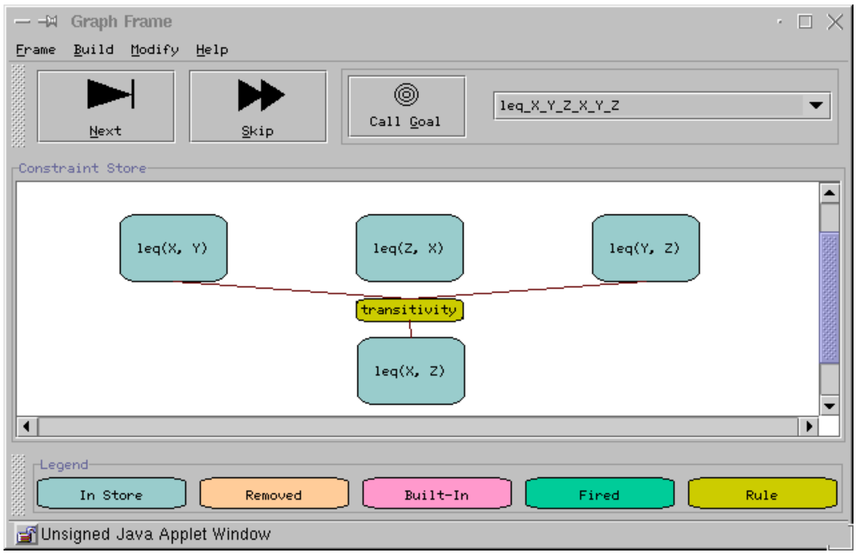


Figure 3.3: Step-by-Step Expansion

Expansion of the graph need not proceed in a step-by-step fashion. The operation "skip" creates the whole graph. Figure 3.4 shows the graph resulting from "skip" activated for the root. To distinguish between user-defined and built-in constraints, we use different colors for them.

In a box view, the sequence initially consists of a root node associated with the initial constraint store, i.e. the goal (Figure 3.1). Application of rules induces a dependency relationship between the constraints in the constraint store. This relationship can be displayed by marking one or more constraints which cause a rule to fire with a different color. Figure 3.5 shows that first `leq(X,Y)` and `leq(Y,Z)` are used to apply the transitivity rule and then `leq(X,Y)` and `leq(Y,Z)` cause the transitivity rule to fire again.

3.3.3 Hiding Nodes

Typical graphs and sequences are too complex to be handled conveniently by the techniques described so far. They require means to change (temporarily) their structure, such that the user sees a compactified version abstracting from details that are currently irrelevant. VisualCHR offers the concept of hiding nodes representing either the constraints or the rules.

The user has a wide range of possibilities to hide and "unhide" nodes explicitly and to specify conditions to hide and unhide nodes automatically. For example, one mouse click is sufficient to transform the sequence representing the execution of the goal `leq(X,Y), leq(Z,X), leq(Y,Z)` such that only a small

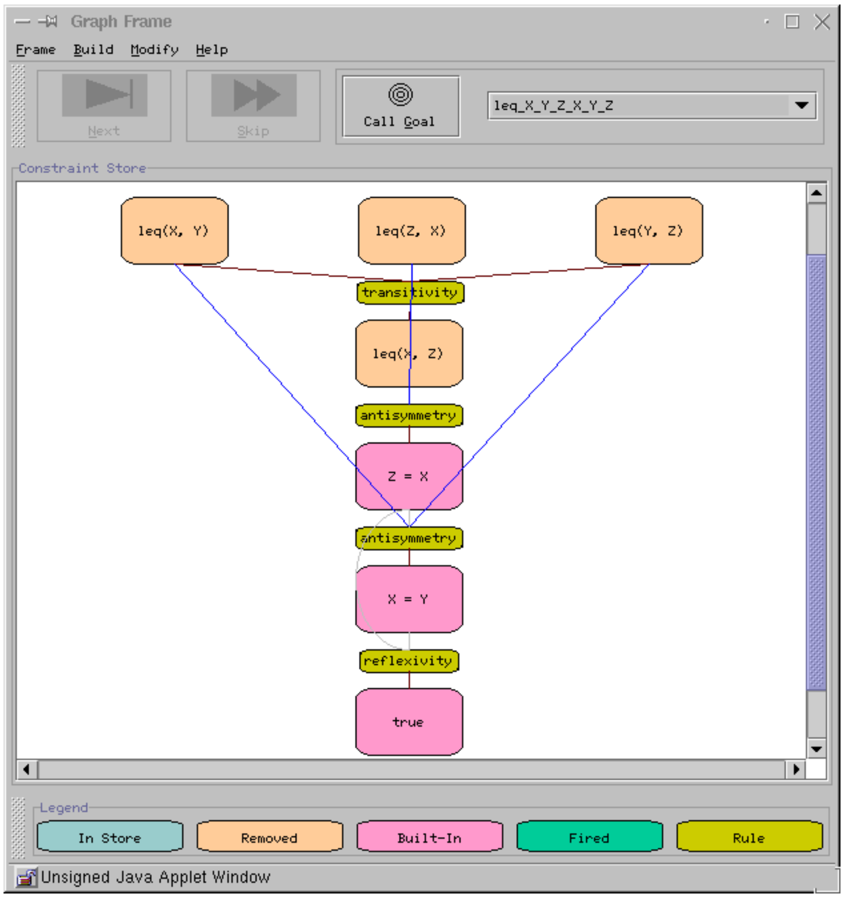


Figure 3.4: Expansion

part of the sequence is displayed, i.e. only the first derivation step and the (four) remaining steps are hidden.

3.3.4 Implementation Issues

VisualCHR is implemented in Java. The implementation is divided into two parts:

- Laying out and drawing the graph. That includes support for scaling the graph, as well as support for hiding and unhiding of nodes.
- The user interface which provides for menus, cursor control, status bar, ...

The user interface is implemented using Swing [40].

The method for computing the layout is still primitive. However, the user has the possibility to interact and to change the layout manually. We first considered to use existing tools for drawing layouts for graphs, e.g. the graph visualization system *daVinci* [76]. Unfortunately, it is hard to design a powerful user interface

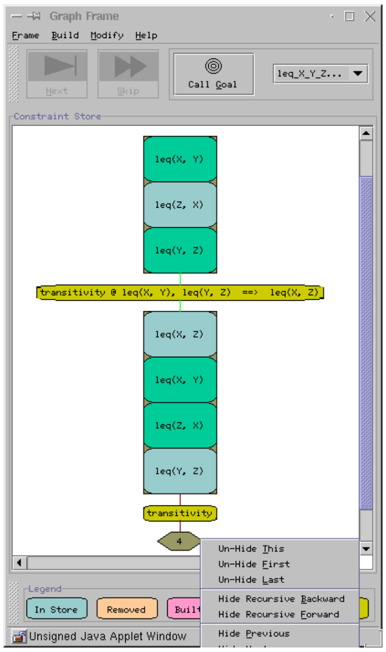


Figure 3.5: Hiding Nodes

since the tools have a user interface of their own which can be customized in a limited fashion only. Nevertheless, the method for computing and drawing the graphs has to be improved by using more sophisticated approaches, e.g. [84].

3.4 Conclusion and Future Work

In Section 3.2, we introduced the notion of operational equivalence of CHR programs. We gave a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent CHR programs. A decidable, sufficient and necessary condition for confluence of a terminating CHR programs was given in earlier work [6, 1, 7]. We have also shown that an extension of the confluence notion to two programs, called compatibility, is not sufficient to capture the operational semantics of CHR programs.

For practical reasons, we also investigated a notion of operational equivalence for user-defined constraints that are defined in different programs. We gave a sufficient syntactic condition for constraints defined in terminating and confluent CHR programs. For a subclass of programs which have only one user-defined constraint in common, we were able to give a sufficient and necessary syntactic condition.

Future work aims to enlarge the class of CHR programs for which we can give a sufficient and necessary syntactic condition for operational equivalence. We also plan to investigate the relationship between operational equivalence and logical equivalence of CHR programs. Furthermore, operational equivalence together with completion [4] provide a good starting point for investigating partial evaluation, and program transformation in general, of constraint solvers.

In Section 3.3, we have presented an interactive tool, called VisualCHR, to visualize the propagation and simplification of constraints. VisualCHR is used to debug and to improve the efficency of constraint solvers written in the high-level language Constraint Handling Rules. It can also be used to understand the details of constraint propagation methods and the interaction of different constraints.

Currently, we are trying to provide a plug-in mechanism for changing the representation of the constraints. For example, cumulative constraints [15] in a scheduling application could be represented as Gantt-charts, reflecting their role in concrete application. VisualCHR is a part of the Java constraint library JACK (Chapter 6). A direction for future work will be the design of an interaction between VisualCHR and a visualization tool for search trees.

Chapter 4

Automatic Generation of Constraint Solvers

A general approach to implement propagation and simplification of constraints consists of applying rules over these constraints. However, a difficulty that arises frequently when writing a constraint solver is to determine the constraint propagation algorithm. In this work, we propose a method to generate automatically the propagation and simplification process of constraints in form of rules. The generated rules are implemented in the language Constraint Handling Rules.

The approach we have taken is to develop an automatic method to generate general rules defining some properties of constraints given their extensional definition. Using our method, the user has the possibility to specify the form of the rules she/he wants to generate. The method allows any kind of constraints in the left hand side of rules and in their right hand side as well. The generation of rules is performed in two steps. In a first step, only propagation rules are generated (Section 4.1) [10]. This method is inspired by techniques used in the field of knowledge discovery. Since a propagation rule does not rewrite constraints but adds new ones, the constraint store may contain superfluous information. Constraints can be removed from the constraint store using simplification rules. In general, removing constraints improves both the time and space behavior of constraint solving. Thus, in a further step we propose a syntactical method to decide when and how to transform propagation rules into simplification rules (Section 4.2) [11]. The method is based on the confluence notion presented in Section 3.1.

Consider the following example, where the user wants to generate a constraint solver for the boolean conjunction $and(X, Y, Z)$, where X and Y are the input arguments and Z is the output argument. This ternary relation can be defined extensionally by the triples $\{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\}$, where 1 stands for truth and 0 for falsity. First, the following propagation rules are generated provided the user specifies their left hand side to be the *and* constraint and their right hand side to be a conjunction of equality constraints:

$$\begin{array}{rcl} and(0,Y,Z) & \Rightarrow & Z{=}0. \\ and(X,0,Z) & \Rightarrow & Z{=}0. \\ and(1,Y,Z) & \Rightarrow & Y{=}Z. \\ and(X,1,Z) & \Rightarrow & X{=}Z. \\ and(X,X,Z) & \Rightarrow & X{=}Z. \\ and(X,Y,1) & \Rightarrow & X{=}1 \wedge Y{=}1. \end{array}$$

For example, the first rule says that the constraint $and(X,Y,Z)$, when it is known that the first input argument X is equal to 0, can propagate the constraint that the output Z must be equal to 0. Hence the goal $and(0,Y,Z)$ will result in $and(0,Y,Z) \wedge Z{=}0$.

In a second step, all propagation rules are transformed into the following simplification rules:

$$\begin{array}{rcl} and(0,Y,Z) & \Leftrightarrow & Z{=}0. \\ and(X,0,Z) & \Leftrightarrow & Z{=}0. \\ and(1,Y,Z) & \Leftrightarrow & Y{=}Z. \\ and(X,1,Z) & \Leftrightarrow & X{=}Z. \\ and(X,X,Z) & \Leftrightarrow & X{=}Z. \\ and(X,Y,1) & \Leftrightarrow & X{=}1 \wedge Y{=}1. \end{array}$$

Now, the first rule says that the constraint $and(0,Y,Z)$ can be replaced by the equality constraint $Z{=}0$. These rules are the well-known rules that can be found in several papers describing the propagation of boolean constraints, e.g. in form of demons [39], conditionals [99], CHR rules [46] or proof systems [36, 21].

This chapter is organized as follows: In section 4.1, we present the algorithm for the generation of propagation rules and give some soundness, correctness and termination results. Then, we give more examples for the use of this algorithm. In section 4.2, we present a syntactical method to transform propagation rules into simplification rules and give some properties of the transformation. In Section 4.3, we present an example to show the practical usefulness of the automatic generation of constraint solvers. In Section 4.4, we compare our approach with already existing work. Finally, we conclude with a summary and directions for future work.

4.1 Generation of Propagation Rules

4.1.1 The PropMiner Algorithm

In this section, we describe the algorithm, PropMiner, for generating propagation rules. This method has been developed based on previous work done in the field of *knowledge discovery*. More precisely, we combine several techniques stemming from two domains: *association rule mining* [17] and Inductive Logic

Programming (ILP) [75]. Note that in the presentation of our algorithm we use an abstract representation of propagation rules. Built-in constraints may appear in the left hand side of these rules in contrast to propagation rules of CHR. We later present how these abstract rules are implemented in CHR.

Class of Generated Rules

A *constraint* over a set of atomic constraints $\mathcal{A}$ is a finite subset of $\mathcal{A}$. A constraint $C \subseteq \mathcal{A}$ is interpreted as the conjunction of the atomic constraints in C. The set of all constraints over $\mathcal{A}$, i.e. the set of all non-empty finite subsets of $\mathcal{A}$, is noted $\mathcal{L}(\mathcal{A})$. The set of variables appearing in $\mathcal{A}$ is denoted by $Var(\mathcal{A})$.

Let CT be a constraint theory defining a constraint C and let σ be a ground substitution. σ is a *solution* of C if and only if $CT \models \sigma(C)$.

A *propagation rule* is a rule of the form $C_1 \rightarrow C_2$, where C_1 and C_2 are constraints. C_1 is called the left hand side (lhs) and C_2 the right hand side (rhs) of the rule.

Definition 4.1 Let $\mathcal{A}_{lhs}$ and $\mathcal{A}_{rhs}$ be two sets of atomic constraints not containing *false*[1]. The set of *propagation rules* over $\langle \mathcal{A}_{lhs}, \mathcal{A}_{rhs} \rangle$ is the set of all rules of the form $C_1 \rightarrow C_2$, where $C_1 \in \mathcal{L}(\mathcal{A}_{lhs})$ and $C_2 \in \mathcal{L}(\mathcal{A}_{rhs}) \cup \{\{false\}\}$ and $C_1 \cap C_2 = \emptyset$. A *failure rule* is a propagation rule of the form $C_1 \rightarrow \{false\}$. □

Definition 4.2 A propagation rule $C_1 \rightarrow C_2$ is *valid* if and only if for any ground substitution σ, if σ is a solution of C_1 then σ is a solution of C_2. The rule $C_1 \rightarrow \{false\}$ is *valid* if and only if C_1 has no solution. □

Since the number of valid rules may become quite large, we considered that the rules that are in some sense the most general will be the most interesting to build a solver. We consider only a syntactical notion of rule generality which is inspired by the notion of structural covering used in association rule mining [95].

Definition 4.3 Let $\mathcal{R}$ and $\mathcal{R}'$ be two sets of propagation rules. $\mathcal{R}'$ is a *cover* of $\mathcal{R}$ if and only if for all $(C_1 \rightarrow C_2) \in \mathcal{R}$ there exists $(C_1' \rightarrow C_2') \in \mathcal{R}'$, such that $C_1' \subseteq C_1$ and $C_2 \subseteq C_2'$. □

Note that this is a form of subsumption in the ground case and that if $\mathcal{R}'$ is a cover of $\mathcal{R}$, then every rule in $\mathcal{R}$ is logically entailed in CT by some rule in $\mathcal{R}'$.

Example 4.4 Let *and* be a ternary constraint defining the Boolean conjunction. $\{\{and(X,Y,Z),\ X{=}0\} \rightarrow \{Z{=}0\}\}$ is a cover of $\{\{and(X,Y,Z),\ X{=}0\} \rightarrow \{Z{=}0\}, \{and(X,Y,Z),\ X{=}0,\ Y{=}0\} \rightarrow \{Z{=}0\}\}$. □

The algorithm PROPMINER generates a cover of the set of *propagation rules* over $\langle \mathcal{A}_{lhs}, \mathcal{A}_{rhs} \rangle$. However, many lhs are of little interest to build solvers based on propagation rules. Then as in ILP [75] we used a syntactic bias to restrict the generation to a particular set of rules called *relevant propagation rules*.

[1] *false* will be used as a particular rhs for the rules.

Definition 4.5 Let $Base_{lhs}$ be a set of atomic constraints. A set of atomic constraints $\mathcal{A}$ is an *interesting pattern* wrt. $Base_{lhs}$ if and only if the following conditions are satisfied:

1. $Base_{lhs} \subseteq \mathcal{A}$.
2. if $|\mathcal{A}| > 1$ then any atomic constraint in $\mathcal{A}$ shares at least one variable with another atomic constraint in $\mathcal{A}$.

□

Definition 4.6 The set of *relevant propagation rules* over $\langle Base_{lhs}, \mathcal{A}_{lhs}, \mathcal{A}_{rhs}\rangle$ is the set of propagation rules over $\langle \mathcal{A}_{lhs}, \mathcal{A}_{rhs}\rangle$ without the rules with a left hand side that is not an interesting pattern wrt. $Base_{lhs}$. □

Example 4.7 Assume we want to generate interaction rules between the Boolean operations conjunction (*and*) and negation (*neg*), then $Base_{lhs}$ has the following form $\{and(X,Y,Z),\ neg(A,B)\}$.
$\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y\} \rightarrow \{Z{=}0\}$ is then a relevant propagation rule, while the rule $\{and(X,Y,Z),\ Y{=}1\} \rightarrow \{Z{=}X\}$ and the rule $\{and(X,Y,Z),\ neg(A,B),\ X{=}0\} \rightarrow \{Z{=}0\}$ are not. However, it should be noticed that the first one will be relevant for the constraint *and* alone (i.e., when $Base_{lhs} = \{and(X,Y,Z)\}$). □

The generation algorithm will discard any rule which is not a relevant propagation rule over a given $\langle Base_{lhs}, \mathcal{A}_{lhs}, \mathcal{A}_{rhs}\rangle$. We present in Section 4.1.3 additional simplifications of the set of rules generated to remove some redundancies.

The Algorithm

Using PropMiner the user has the possibility to specify the admissible syntactic forms of the rules. The user determines the constraint for which rules have to be generated (i.e. $Base_{lhs}$) and chooses the candidate constraints to form conjunctions together with $Base_{lhs}$ in the left hand side (noted $Cand_{lhs}$). Usually, these candidate constraints are simply equality constraints. For the right hand side of the rules the user specifies also the form of candidate constraints she/he wants to see there (noted $Cand_{rhs}$). Finally, the user determines the semantics of the constraint $Base_{lhs}$ by means of its extensional definition (noted $SolBase_{lhs}$) which must be finite, and provides the semantics of the candidate constraints $Cand_{lhs}$ and $Cand_{rhs}$ by two constraint theories CT_{lhs} and CT_{rhs}, respectively. Furthermore, we assume that the constraints defined by CT_{lhs} and CT_{rhs} are handled by an appropriate constraint solver.

To compute the propagation rules the algorithm generates each possible lhs constraint (noted C_{lhs}) and for each determines the corresponding rhs constraint (noted C_{rhs}).

For each lhs C_{lhs} the corresponding rhs C_{rhs} is computed in the following way:

1. if C_{lhs} has no solution then $C_{rhs} = \{false\}$ and we have the failure rule $C_{lhs} \rightarrow \{false\}$.

2. if C_{lhs} has at least one solution then C_{rhs} is the set of all atomic constraints that are candidates for the rhs part and are true for all solutions of C_{lhs}. If C_{rhs} is not empty we have the rule $C_{lhs} \to C_{rhs}$.

During the exploration of the search space, the algorithm uses two main pruning strategies:

1. (*Pruning1*) if a rule $C_{lhs} \to \{false\}$ is generated then there is no need to consider any superset of C_{lhs} to form other rule lhs.

2. (*Pruning2*) if a rule $C_{lhs} \to C_{rhs}$ is generated then there is no need to consider any C such that $C_{lhs} \subset C$ and $C \cap C_{rhs} \neq \emptyset$ to form other rule lhs.

The condition $C \cap C_{rhs} \neq \emptyset$ in the strategy *Pruning2* is needed to reduce the number of the propagation rules generated, as shown in the following example.

Example 4.8 After generating the relevant propagation rule of example 4.7: $\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y\} \to \{Z{=}0\}$, the possible lhs $\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y,\ B{=}1,\ Z{=}0\}$ is not considered using *Pruning2*, while $\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y,\ B{=}1\}$ remains a lhs candidate and may lead to the following rule
$\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y,\ B{=}1\} \to \{Z{=}0,\ A{=}0,\ X{=}0,\ Y{=}1\}$. □

These pruning strategies are much more efficient if during the enumeration of all possible rule lhs, a given lhs is considered before any of its supersets. So a specific ordering for this enumeration is imposed in the algorithm. Moreover, this ordering allows to discover early covering rules avoiding then the generation of many uninteresting covered rules.

To simplify the presentation of the algorithm we consider that all possible lhs are stored in a list L and that unnecessary lhs candidates are simply removed from this list. For efficiency reasons the concrete implementation is not based on a list but on a tree containing lhs candidates on its nodes. More details are given in Section 4.1.4.

We now give an abstract description of the PropMiner algorithm. It takes as input:

- $Base_{lhs}$: a constraint that must be included in any lhs of the rules.

- $SolBase_{lhs}$: the finite set of ground substitutions that are solutions of $Base_{lhs}$. Note that this defines the constraint $Base_{lhs}$ extensionally.

- $Cand_{lhs}$: a finite set of atomic constraints that are candidates to form lhs of the rules such that $Var(Cand_{lhs}) \subseteq Var(Base_{lhs})$.

- $Cand_{rhs}$: a finite set of atomic constraints that are candidates to form rhs of the rules such that $Var(Cand_{rhs}) \subseteq Var(Base_{lhs})$.

- CT_{lhs}: a constraint theory defining $Cand_{lhs}$.

- CT_{rhs}: a constraint theory defining $Cand_{rhs}$.

And it produces the following output:

- a cover of the valid relevant propagation rules over $\langle Base_{lhs}, Cand_{lhs}, Cand_{rhs}\rangle$

PropMiner **Algorithm**

begin

Let $\mathcal{R}$ be an empty set of rules.
Let L be a list containing the elements of $\mathcal{L}(Base_{lhs} \cup Cand_{lhs})$ in any order.

Remove from L any element which is not an interesting pattern wrt. $Base_{lhs}$.
Order L with any total ordering compatible with the subset partial ordering (i.e., for all C_1 in L if C_2 is after C_1 in L then $C_2 \not\subset C_1$).

while L is not empty **do**
  Let C_{lhs} be the first element of L.
  Remove from L its first element.
    if for all $\sigma \in SolBase_{lhs}$ we have
    $CT_{lhs} \models \neg\sigma(C_{lhs} \setminus Base_{lhs})$ **then**
      add the failure rule $(C_{lhs} \rightarrow \{false\})$ to $\mathcal{R}$
      and remove from L each element C such that $C_{lhs} \subset C$.
    else
      compute C_{rhs} the rule rhs, defined by
      $C_{rhs} = \{c | c \in (Cand_{rhs} \setminus Cand_{lhs})$ and for all $\sigma \in SolBase_{lhs}$
        when $CT_{lhs} \models \sigma(C_{lhs} \setminus Base_{lhs})$ we have $CT_{rhs} \models \sigma(c)\}$.
      if C_{rhs} is not empty **then**
        add the rule $(C_{lhs} \rightarrow C_{rhs})$ to $\mathcal{R}$
        and remove from L each element C such that
          $C_{lhs} \subset C$ and $C \cap C_{rhs} \neq \emptyset$.
      endif
    endif
endwhile

output $\mathcal{R}$

end

We require the constraint theories CT_{lhs} and CT_{rhs} to be ground complete for $\langle Cand_{lhs}, SolBase_{lhs}\rangle$ and $\langle Cand_{rhs}, SolBase_{lhs}\rangle$, respectively[2].

Definition 4.9 Let CT be a constraint theory, let Γ be a set of ground substitutions and $\mathcal{A}$ be a set of atomic constraints. CT is *ground complete* for $\langle \mathcal{A}, \Gamma\rangle$ if and only if for every $c \in \mathcal{A}$ and for any substitution $\sigma \in \Gamma$ we have either $CT \models \sigma(c)$ or $CT \models \neg\sigma(c)$.

□

[2]Note that this restriction is very weak, since the property holds for almost all useful classes of constraint theories.

4.1.2 Properties of the PropMiner Algorithm

In PropMiner the list L of possible lhs is initialized to be a finite list. Each iteration of the while loop removes at least one element in L. This ensures the following property.

Theorem 4.10 (Termination) The algorithm PropMiner terminates and yields a finite set of propagation rules. □

The following results establish soundness and correctness of the algorithm.

Theorem 4.11 (Soundness) PropMiner computes valid relevant propagation rules over $\langle Base_{lhs}, Cand_{lhs}, Cand_{rhs}\rangle$. **Proof:** All C_{lhs} considered are interesting pattern wrt. $Base_{lhs}$, thus only relevant rules can be generated. Let $C'_{lhs} = C_{lhs} \setminus Base_{lhs}$. A rule of the form $C_{lhs} \rightarrow \{false\}$ can be generated only if all solutions of $Base_{lhs}$ are not solutions of C'_{lhs}. So any rule $C_{lhs} \rightarrow \{false\}$ generated is valid. A rule of the form $C_{lhs} \rightarrow C_{rhs}$, where $C_{rhs} \neq \{false\}$ can be generated only if all solutions of $Base_{lhs}$ that are solutions of C'_{lhs}, are also solutions of all atomic constraints in C_{rhs}. Hence all generated rules of the form $C_{lhs} \rightarrow C_{rhs}$ are valid.

□

Theorem 4.12 (Correctness) PropMiner computes a cover of the valid relevant propagation rules over $\langle Base_{lhs}, Cand_{lhs}, Cand_{rhs}\rangle$ when $Var(Cand_{lhs}) \subseteq Var(Base_{lhs})$ and $Var(Cand_{rhs}) \subseteq Var(Base_{lhs})$.

Proof: First, we do not consider the two pruning strategies *Pruning1* and *Pruning2*. Then the algorithm enumerates all possible rule lhs that are interesting pattern wrt. $Base_{lhs}$. So it generates all valid relevant failure rules. Moreover for any valid relevant rule of the form $C_1 \rightarrow C_2$, where $C_2 \neq \{false\}$ the algorithm considers C_1 as a candidate lhs. Then it computes C_{rhs} containing all atomic constraints c such that all solutions of C_1 are solutions of c. Thus $C_2 \subseteq C_{rhs}$. So if we do not consider the two pruning strategies *Pruning1* and *Pruning2* the algorithm outputs a cover of the valid relevant propagation rules.

Now we show that the two pruning criteria are safe.

(*Pruning1*) When a rule of the form $C_1 \rightarrow \{false\}$ is generated all candidate lhs $C_{lhs} \supset C_1$ are discarded. However since $C_1 \rightarrow \{false\}$ is valid, C_1 has no solution, and thus any $C_{lhs} \supset C_1$ have no solution too, and can only lead to a rule of the form $C_{lhs} \rightarrow \{false\}$ which will be covered by $C_1 \rightarrow \{false\}$.

(*Pruning2*) When a rule of the form $C_1 \rightarrow C_2$, where $C_2 \neq \{false\}$ is generated all candidates lhs C_{lhs} such that $C_1 \subset C_{lhs}$ and $C_{lhs} \cap C_2 \neq \emptyset$ are discarded.

The key idea of the safety of this pruning criterion is the following:

Consider a C_{lhs} discarded and any valid relevant rule of the form $C_{lhs} \rightarrow C_3$, where $Var(C_3) \subseteq Var(C_{lhs})$. There is another lhs candidate $C'_{lhs} = C_{lhs} \setminus C_2$ that has not been discarded, such that $C_1 \subseteq C'_{lhs}$. Any ground substitution σ solution of C'_{lhs} is a solution of C_1 and thus of C_2 because $C_1 \rightarrow C_2$ is valid. Then σ is a solution of C_{lhs} and also of C_3 since $C_{lhs} \rightarrow C_3$ is valid. Hence the candidate C'_{lhs} will generate a rule $C'_{lhs} \rightarrow C_4$ with $C_3 \subseteq C_4$ that covers rule $C_{lhs} \rightarrow C_3$. □

4.1.3 Rule Simplification

Even though PropMiner computes a cover of the valid relevant propagation rules over $\langle Base_{lhs}, Cand_{lhs}, Cand_{rhs}\rangle$, this cover may contain some kind of redundancies.

Example 4.13 PropMiner as presented above can produce the following rule $\{and(X,Y,Z),\ Z{=}1\} \rightarrow \{X{=}Y,\ X{=}Z,\ Y{=}Z,\ X{=}1,\ Y{=}1\}$. If we have already a solver to handle equality constraints, then this rule can be simplified into $\{and(X,Y,1)\} \rightarrow \{X{=}1,\ Y{=}1\}$.
On another input PropMiner can produce the following rules for the logical operation exclusive-or (xor): $\{xor(X,Y,Z),\ X{=}Y\} \rightarrow \{Z{=}0\}$ and the rule $\{xor(X,Y,Z),\ X{=}0,\ Y{=}0\} \rightarrow \{Z{=}0\}$. The second rule cannot propagate new atomic constraints wrt. the first rule, and thus can be discarded. □

We use an ad-hoc technique to simplify the rule lhs and rhs, and to suppress some redundant rules. This process does not lead to a precisely defined canonical representation of the rules generated, but in practice (see Section 4.1.5) it produces small and readable sets of rules.

This simplification technique is incorporated in PropMiner and performed during the generation of the rules. For clarity reasons it is presented apart from the algorithm given above. The simplification principle is as follows:

- For each rule generated by the PropMiner algorithm the equality constraints appearing in the lhs are transformed into substitutions that are applied to the lhs and the rhs, and then the completely ground atomic constraints are removed from lhs and rhs (e.g., $\{and(X,Y,Z),\ Z{=}1\} \rightarrow \{X{=}Y,\ X{=}Z,\ Y{=}Z,\ X{=}1,\ Y{=}1\}$ will be simplified to the propagation rule $\{and(X,Y,Z),\ Z{=}1\} \rightarrow \{X{=}Y,\ X{=}1,\ Y{=}1\}$).

- The new rules are then ordered in a list L' using any total ordering on the rule lhs compatible with the θ-subsumption ordering [79] (i.e., a rule having a more general lhs is placed before a rule with a more specialized lhs).

- Let S be a set of rules initialized to the empty set. For each rule $C_1 \rightarrow C_2$ in L' (taken according to the list ordering) the constraint C_2 will be simplified to an equivalent constraint C_{simp} by the already known solver for $Cand_{rhs}$ and by the rules in S. If C_{simp} is empty then the rule can be discarded, else add the rule $C_1 \rightarrow C_{simp}$ to S. (e.g., $\{and(X,Y,Z),\ Z{=}1\} \rightarrow \{X{=}Y,\ X{=}1,\ Y{=}1\}$ will be simplified to the rule $\{and(X,Y,Z),\ Z{=}1\} \rightarrow \{X{=}1,\ Y{=}1\}$)

- Output the set S containing the simplified set of rules.

In contrast to the propagation rules of CHR built-in constraints may appear in the left hand side of the rules generated by the PropMiner algorithm. Therefore, after the simplification process presented above, the resulting rules are transformed into propagation rules of CHR. Usually, the candidate constraints $Cand_{lhs}$ are simply equality constraints. Thus, equality constraints appearing in the lhs of a rule are propagated all over the constraints $Base_{lhs}$ (e.g., $\{and(X,Y,Z),\ Z{=}1\} \rightarrow \{X{=}1,\ Y{=}1\}$ will be transformed into

$and(X, Y, 1) \Rightarrow X{=}1 \wedge Y{=}1$. For other built-in constraints the transformation leads to CHR rules consisting of a guard [46] (e.g. $\{and(X, Y, Z),\ Z{\neq}0\} \rightarrow \{X{=}1,\ Y{=}1\}$ will be transformed to the CHR propagation rule $and(X, Y, Z) \Rightarrow Z{\neq}0 \mid X{=}1 \wedge Y{=}1$).

4.1.4 Implementation Issues

As described in Section 4.1.1, the PropMiner algorithm needs to enumerate lhs constraints. Our implementation follows the idea of direct extraction of association rules by exploring a tree corresponding to the lhs search space as described in [23]. This tree is expanded and explored using a depth first strategy, in a way that constructs only necessary lhs candidates and allows to remove uninteresting candidates by cutting whole branches of the tree. The branches of the tree are developed using a partial ordering on the lhs candidates such that the more general lhs are examined before more specialized ones. The partial ordering used in our implementation is the θ-subsumption [79] ordering commonly used in ILP to structure the search space [75]. To prune branches in the tree, one of the two main strategies (*Pruning2*) has been inspired by the Close algorithm [77] devoted to the extraction of frequent *itemsets* in dense[3] data sets.

The running prototype is implemented in SICStus Prolog 3.7.1 and takes advantage of the support of CHR in this environment in the following way. During the execution of the PropMiner algorithm we build incrementally a CHR solver with the propagation rules generated and this solver is used to perform the rule simplification according to Section 4.1.3.

Example 4.14 Let $and(X, Y, 1) \Rightarrow X{=}1 \wedge Y{=}1$ be a rule in the current CHR solver. Then the rule $and(0, Y, 1) \Rightarrow \mathit{false}$ will be discarded since $and(0, Y, 1)$ leads already to a failure using the current solver. □

4.1.5 Examples

This section shows with examples that a practical application of our method lies in software development. The rules generated by the PropMiner algorithm will be implemented as propagation rules of the language Constraint Handling Rules. While we cannot – within the space limitations – introduce the whole generated constraint solver, we still give a fragment of it. The complete solvers are available in [33] and can be executed online.

In the following, we assume that the constraint theories define among other constraints equality ("=") and disequality ("≠") as syntactic equality and disequality. Furthermore, we assume that these constraints are handled by an appropriate constraint solver.

For convenience, we introduce the following notation. Let c be a constraint symbol of arity 2 and D_1 and D_2 be two sets of terms. We define $atomic(c, D_1, D_2)$ as the set of all atomic constraints built from c over $D_1 \times D_2$. More precisely, $atomic(c, D_1, D_2) = \{c(\alpha, \beta) \mid \alpha \in D_1 \text{ and } \beta \in D_2\}$.

Other automatically generated propagation rules are described in [20, 83]. We will compare our results only to the ones presented in [20] (See Section 4.4

[3] e.g., data sets containing many strong correlations.

for more details), since experiments with practical applications are still missing in [83]. One can remark that the times for the generation of rules are in the same order of magnitude. We have used the following software and hardware: SICStus Prolog 3.7.1, PC Pentium 3 with 256 MBytes of memory and a 500 MHZ processor.

Boolean Constraints

Boolean primitive constraints consist of Boolean variables which may take the value `0` for falsity or `1` for truth, and Boolean operations such as conjunction (`and`), disjunction (`or`), negation (`neg`) and exclusive-or (`xor`), modeled here as relations.

For the conjunction constraint $and(X, Y, Z)$ the algorithm PropMiner with the following input

$$\begin{aligned} Base_{lhs} &= \{and(X, Y, Z)\} \\ Cand_{lhs} &= Cand_{rhs} = atomic(=, \{X, Y, Z\}, \{X, Y, Z, 0, 1\}) \end{aligned}$$

generates the following rules in 0.05 seconds:

$$\begin{aligned} and(0, Y, Z) &\Rightarrow Z{=}0. \\ and(X, 0, Z) &\Rightarrow Z{=}0. \\ and(1, Y, Z) &\Rightarrow Y{=}Z. \\ and(X, 1, Z) &\Rightarrow X{=}Z. \\ and(X, X, Z) &\Rightarrow X{=}Z. \\ and(X, Y, 1) &\Rightarrow X{=}1 \wedge Y{=}1. \end{aligned}$$

Goals of the form $and(X, X, Z)$ cannot be handled using the rules generated by the algorithm presented in [20], since the second last rule is not present there. One can easily see that the propagation rules generated by the PropMiner algorithm correspond to the implementation of *and* using CHR [46].

For the negation constraint $neg(X, Y)$, the PropMiner algorithm generates among other rules the following failure rule:

$$neg(X, X) \Rightarrow false.$$

The algorithm PropMiner can generate propagation rules defining interactions between constraints. With the following input

$$\begin{aligned} Base_{lhs} &= \{and(X, Y, Z), neg(A, B)\} \\ Cand_{lhs} &= Cand_{rhs} = atomic(=, \{X, Y, Z, A, B\}, \{X, Y, Z, A, B, 0, 1\}) \end{aligned}$$

the following rules defining interaction between *neg* and *and* are generated:

$$\begin{aligned} and(X, Y, Z) \wedge neg(X, Y) &\Rightarrow Z{=}0. \\ and(X, Y, Z) \wedge neg(Y, X) &\Rightarrow Z{=}0. \\ and(X, Y, Z) \wedge neg(X, Z) &\Rightarrow X{=}1 \wedge Y{=}0 \wedge Z{=}0. \\ and(X, Y, Z) \wedge neg(Z, X) &\Rightarrow X{=}1 \wedge Y{=}0 \wedge Z{=}0. \\ and(X, Y, Z) \wedge neg(Y, Z) &\Rightarrow X{=}0 \wedge Y{=}1 \wedge Z{=}0. \\ and(X, Y, Z) \wedge neg(Z, Y) &\Rightarrow X{=}0 \wedge Y{=}1 \wedge Z{=}0. \end{aligned}$$

With our algorithm, propagation rules with a right hand side consisting of more complex constraints than equality constraints can also be generated. The user can specify the form of the right hand side of the rules. Using the PropMiner algorithm with the following input

$$\begin{aligned} Base_{lhs} &= \{xor(X,Y,Z)\} \\ Cand_{lhs} &= atomic(=, \{X,Y,Z\}, \{X,Y,Z,0,1\}) \\ Cand_{rhs} &= Cand_{lhs} \cup atomic(neg, \{X,Y,Z\}, \{X,Y,Z,0,1\}) \end{aligned}$$

6 rules, analogous to the ones for and, and the following 3 rules for the constraint xor are generated in 0.1 seconds:

$$\begin{aligned} xor(X,Y,1) &\Rightarrow neg(X,Y). \\ xor(X,1,Z) &\Rightarrow neg(X,Z). \\ xor(1,Y,Z) &\Rightarrow neg(Y,Z). \end{aligned}$$

Full-adder

One important application of Boolean constraints is for modeling logic circuits. A full-adder can be built using the following logical gates (see, e.g. [98, 46]):

$$\begin{aligned} & fulladder(X,Y,Z,S,C) \Leftrightarrow \\ & \quad and(X,Y,C1) \wedge \\ & \quad xor(X,Y,S1) \wedge \\ & \quad and(Z,S1,C2) \wedge \\ & \quad xor(Z,S1,S) \wedge \\ & \quad or(C1,C2,C). \end{aligned}$$

Using the PropMiner algorithm 28 rules within 0.68 seconds are generated for the $fulladder$ constraint. These rules enforce the same local consistency notion as the 52 rules generated by the algorithm presented in [20] within 0.27 seconds. Typical rules are:

$$\begin{aligned} fulladder(X,Y,CI,S,S) &\Rightarrow X{=}S \wedge Y{=}S \wedge CI{=}S. \\ fulladder(X,Y,CI,CI,C) &\Rightarrow X{=}C \wedge Y{=}C. \\ fulladder(0,Y,CI,S,1) &\Rightarrow S = 0. \end{aligned}$$

The first rule says that the constraint $fulladder(X,Y,CI,S,C)$, whenever the output bit S is equal to the output carry bit C, can propagate the information that the bits to be added X and Y and the input carry bit CI are equal to the output bit S. This kind of propagation cannot be performed using the rules generated by the algorithm presented in [20].

Three Valued Logics

We consider the equivalence relation defined by the truth table given in [61], where the value t stands for true, f for false and u for unknown.

X	Y	X ≡ Y
t	t	t
t	f	f
t	u	u
f	t	f
f	f	t
f	u	u
u	t	u
u	f	u
u	u	u

The PropMiner algorithm generates for the ternary equivalence constraint *eq3val* 16 rules within 0.3 seconds, e.g.

$$\begin{aligned} eq3val(X,X,X) &\Rightarrow X{\neq}f. \\ eq3val(X,Y,t) &\Rightarrow X{\neq}u \wedge X{=}Y. \\ eq3val(X,f,X) &\Rightarrow X{=}u. \end{aligned}$$

The first rule says that the constraint $eq3val(X,Y,Z)$, when it is known that the input arguments X and Y and the output Z are equal, can propagate that X is different from f.

Temporal Reasoning

In [18] an interval-based approach to temporal reasoning is presented. Allen's approach to reasoning about time is based on the notion of time intervals and binary relations on them. Given two time intervals, their relative positions can be described by exactly one of thirteen primitive interval relations, where each primitive relation can be defined in terms of its endpoint relations, i.e. equality and 6 other relations (*before*, *during*, *overlaps*, *meets*, *starts* and *finishes*) with their converses.

In [18] different ordering relations between intervals are introduced. There is a 13×13 table defining a "composition" constraint between a triple of events X, Y and Z, e.g. if the temporal relations between the events X and Y and the events Y and Z are known, what is the temporal relation between X and Z. The composition constraint, denoted by *allenComp*, can be defined as follows:

$$allenComp(R_1, R_2, R_3) \leftrightarrow (R_1(X,Y) \wedge R_2(Y,Z) \rightarrow R_3(X,Z)),$$

where R_1, R_2, R_3 are primitive interval relations.

Our algorithm generates for *allenComp* 489 rules within 83.12 seconds, provided the user specifies that the right hand side of the rules may consist of a conjunction of equality and disequality constraints. Analogous to [20] we denote the 6 relations respectively by b, d, o, m, s, f, their converses by bi, di, oi, mi, si, fi and the equality relation by e. Typical rules are:

$$\begin{aligned} allenComp(X,X,X) &\Rightarrow X{\neq}m \wedge X{\neq}mi. \\ allenComp(X,X,e) &\Rightarrow X{=}e. \\ allenComp(o,b,Z) &\Rightarrow Z{=}b. \end{aligned}$$

The algorithm presented in [20] generates 498 rules within 31.16 seconds. The right hand side of rules consists only of disequality constraints. Furthermore, rules with multiple occurrences of variables cannot be generated. Thus, no information can be propagated from a constraint of the form $allenComp(X, X, X)$.

Spatial Reasoning

The Region Connection Calculus (RCC) is a topological approach to qualitative spatial representation and reasoning where spatial regions are subsets of topological space [81]. Relationships between spatial regions are defined in terms of the relation $C(X, Y)$ which is true iff the topological closures of regions X and Y share at least one point.

In [81] a composition table for the set of eight basic relations is presented. The relations are *dc* (*disconnected*),*ec* (*externally connected*), *po* (*partial overlap*), *eq* (*equal*), *tpp* (*tangential proper part*) and *ntpp* (*nontangential proper part*). The relations *tpp* and *ntpp* have inverses (here symbolized by *tppi* and *ntppi*).

For the composition constraint $rccComp$, our algorithm generates 178 rules in 24 seconds. Examples of rules are:

$$\begin{array}{rcl} rccComp(X, X, eq) & \Rightarrow & X{\neq}ntppi \wedge X{\neq}tppi \wedge X{\neq}ntpp, X{\neq}tpp. \\ rccComp(X, eq, Z) & \Rightarrow & X{=}Z. \\ rccCom(dc, Y, ntppi) & \Rightarrow & Y{=}dc. \end{array}$$

4.2 Generation of Simplification Rules

Since a propagation rule does not rewrite constraints but adds new ones, the constraint store may contain superfluous information. Constraints can be removed from the constraint store using simplification rules. In general, removing constraints improves both the time and space behavior of constraint solving.

Often generated propagation rules can be rewritten as simplification rules. For the boolean conjunction all propagation rules could be transformed into the following simplification rules:

$$\begin{array}{rcl} and(0, Y, Z) & \Leftrightarrow & Z{=}0. \\ and(X, 0, Z) & \Leftrightarrow & Z{=}0. \\ and(1, Y, Z) & \Leftrightarrow & Y{=}Z. \\ and(X, 1, Z) & \Leftrightarrow & X{=}Z. \\ and(X, X, Z) & \Leftrightarrow & X{=}Z. \\ and(X, Y, 1) & \Leftrightarrow & X{=}1 \wedge Y{=}1. \end{array}$$

Thus, our aim is to find some criteria to perform such a transformation. One simple criterion could be the following: whenever a right hand side of a rule propagates information making its left hand side ground, then this rule can be implemented by means of a simplification rule. This criterion is not sufficient since it can only be applied to the last rule of the example presented above.

Finding a general criterion is even more difficult, when we consider multi-headed propagation rules defining interaction between constraints, i.e. rules with left hand sides consisting of a conjunction of constraints. For example, for the negation constraint $neg(X, Y)$ defined by $\{(0, 1), (1, 0)\}$ and the conjunction constraint $and(X, Y, Z)$ our algorithm generates among other rules the following propagation rules:

$$and(X, Y, Z) \wedge neg(X, Z) \Rightarrow X{=}1 \wedge Y{=}0 \wedge Z{=}0.$$
$$and(X, Y, Z) \wedge neg(Y, Z) \Rightarrow X{=}0 \wedge Y{=}1 \wedge Z{=}0.$$
$$and(X, Y, Z) \wedge neg(X, Y) \Rightarrow Z{=}0.$$

It is obvious that the first two rules can be transformed into simplification rules. However, transforming the third rule to a simplification rule by simply replacing $\Rightarrow$ by $\Leftrightarrow$ leads to losing information about X and Y. One has to keep the negation $neg(X, Y)$ in the constraint store, but remove the $and(X, Y, Z)$:

$$and(X, Y, Z) \wedge neg(X, Y) \quad \Leftrightarrow \quad neg(X, Y) \wedge Z{=}0.$$

In the following, we propose a syntactical method to decide when to transform propagation rules into simplification rules. The method is based on the confluence notion. The idea of the transformation method is to test the confluence of the resulting constraint solver after transforming a propagation rule into a simplification rule. If the resulting solver remains confluent, we can conclude that the transformation leads to an operationally equivalent constraint solver with respect to the built-in constraints.

4.2.1 The SimpMiner Algorithm

We now introduce the algorithm SimpMiner that given a set of propagation rules P generates a set of both propagation and simplification rules. We assume that the CHR program consisting of these propagation rules is terminating. We can easily show that this program is confluent, since propagation rules do not rewrite constraints and adding constraints to a state cannot inhibit the application of a rule as long as the built-in constraints remain consistent. The algorithm works by repeatedly selecting a rule R from P. Then, we try to transform R to a simplification rule. Note that for a propagation rule with multiple heads, there are several possibilities to transform it to a simplification rule. If the resulting program remains confluent then the transformation is accepted and the next rule will be considered. Otherwise, the next transformation of R is tried.

Before giving an abstract description of the SimpMiner algorithm, we illustrate it by the following example:

Example 4.15 Let P be the set of propagation rules defining the negation and the conjunction constraints and the interaction between them.

All single-headed propagation rules can be transformed to simplification rules and the resulting program remains confluent. Now, let R be the propagation rule $and(X, Y, Z) \wedge neg(X, Y) \Rightarrow Z{=}0$.

There are three possibilities to transform R into a simplification rule. If we transform R into the simplification rule $and(X, Y, Z) \wedge neg(X, Y) \Leftrightarrow Z{=}0$, then the resulting program becomes non-confluent since the critical pair

$$(S_1,\ \ S_2) := (Z{=}0,\ \ X{=}Z \wedge neg(X, X))$$

stemming from the critical ancestor state $neg(X,X) \wedge and(X,X,Z)$[4] of the rules $and(X,Y,Z) \wedge neg(X,Y) \Leftrightarrow Z{=}0$ and $and(X,X,Z) \Leftrightarrow X{=}Z$ is not joinable, i.e. S_1 is a final state and S_2 leads to false.

If we transform the propagation rule R into a simplification rule of the form $and(X,Y,Z) \wedge neg(X,Y) \Leftrightarrow and(X,Y,Z) \wedge Z{=}0$, then the resulting program becomes also non-confluent.

However, transforming the propagation rule R into a simplification rule of the form $and(X,Y,Z) \wedge neg(X,Y) \Leftrightarrow neg(X,Y) \wedge Z{=}0$ leads to a confluent program. Thus, this transformation is accepted and we proceed with the next propagation rule. Finally, the transformed program consists of the single-headed simplification rules defining the conjunction and negation constraints and of the following rules defining their interaction:

$$\begin{array}{lcl}
and(X,Y,Z) \wedge neg(X,Y) & \Leftrightarrow & neg(X,Y) \wedge Z{=}0. \\
and(X,Y,Z) \wedge neg(Y,X) & \Leftrightarrow & neg(Y,X) \wedge Z{=}0. \\
and(X,Y,Z) \wedge neg(X,Z) & \Leftrightarrow & X{=}1 \wedge Y{=}0 \wedge Z{=}0. \\
and(X,Y,Z) \wedge neg(Z,X) & \Leftrightarrow & X{=}1 \wedge Y{=}0 \wedge Z{=}0. \\
and(X,Y,Z) \wedge neg(Y,Z) & \Leftrightarrow & X{=}0 \wedge Y{=}1 \wedge Z{=}0. \\
and(X,Y,Z) \wedge neg(Z,Y) & \Leftrightarrow & X{=}0 \wedge Y{=}1 \wedge Z{=}0.
\end{array}$$

These rules are slightly different from the ones implemented manually [33]. However, one can easily show that the two programs are operationally equivalent (Section 3.2).

□

We now give an abstract description of the SimpMiner algorithm.

SimpMiner **Algorithm**

INPUT: A set of propagation rules P.
OUTPUT: A built-in operationally equivalent guard-free CHR program P' consisting of propagation and simplification rules.
ALGORITHM:

$P' := P$
for each rule R of the form $H \Rightarrow B$ in P **do**
  Find $R' := H \Leftrightarrow B \wedge C$, where C is a subconjunction of H such that $(P' \backslash \{R\}) \cup \{R'\}$ is terminating and all its critical pairs are joinable.
    If R' exists
    then $P' := (P' \backslash \{R\}) \cup \{R'\}$
    endif
endfor
return P'

In the SimpMiner algorithm, there are two degrees of nondeterminism: the choice of a rule R from P and the choice of the subconjunction C of H. In our implementation, the input CHR program is given in a text file and we take the rules according to the order in this file. To handle the second source of nondeterminism and to achieve a form of minimality based on the number

[4] With variables from different rules already identified to have an overlap; for readability.

of constraints, we generate simplification rules that will remove the greatest number of constraints. So, when we try to transform a propagation rule R into a simplification rule R' of the form $H \Leftrightarrow B \wedge C$ we choose the smallest conjunction of constraints C (wrt. the number of constraints in C) for which the resulting program remains confluent, i.e. terminating and all its critical pairs are joinable. If such a C is not unique, we choose any one among the smallest conjunctions. Note that transforming a propagation rule into a simplification rule just by shifting the whole head of the propagation rule into the body of the simplification rule is not allowed since the resulting program becomes non-terminating.

4.2.2 Properties of the SimpMiner Algorithm

In the following, we prove that the transformed set of rules is *built-in-operationally equivalent* to the initial set of propagation rules. Two programs are built-in-operationally equivalent if for each goal, the built-in constraints in the final state in one program are the same as in the final state in the other program. In the following, we will denote the built-in constraints appearing in a state S by $builtIn(S)$.

Theorem 4.16 Let P be a set of propagation rules and P' be the transformed program using the SimpMiner algorithm. If $P \cup P'$ is terminating then for any state S, we have if $S \mapsto^*_{P'} S_1$ and $S \mapsto^*_P S_2$, where S_1 and S_2 are final states, then $builtIn(S_1)$ and $builtIn(S_2)$ are variants.
Proof: A direct consequence of Lemma 4.18 and Lemma 4.19. □

To prove the theorem above, we first show that we can just take the union of the set of propagation rules and the transformed program and the resulting program will be built-in operationally equivalent to the set of propagation rules (cf. Lemma 4.18). Then, we prove that the union of the two programs and the transformed program have the same behavior (cf. Lemma 4.19). Therefore, we can conclude that the transformed program and the initial set of propagation rules are built-in operationally equivalent. To prove Lemma 4.18, we have to show that the union of a set of propagation rules and its transformed program is confluent.

Lemma 4.17 Let P be a set of propagation rules and P' be the transformed program using the SimpMiner algorithm. If $P \cup P'$ is terminating, then $P \cup P'$ is confluent.
Proof: To show that $P \cup P'$ is confluent, we have to show that all critical pairs of $P \cup P'$ are joinable, since $P \cup P'$ is terminating. The set of critical pairs of $P \cup P'$ consists of all critical pairs stemming from two rules appearing in P, all critical pairs stemming from two rules appearing in P' and all critical pairs stemming from one rule appearing in P and one rule appearing in P'.

1) P consists of a set of propagation rules. We can easily show that P is confluent, since any critical pair (S_1, S_2) stemming from two propagation rules R_1 and R_2 is joinable: Let S_i be a state resulting from the application of R_i on the critical ancestor state of R_1 and R_2 $(i = 1, 2)$, then applying R_1 on S_2 and R_2 on S_1 leads to a common state. A more detailed proof can be found in [1].

2) P' is the transformed program using the SIMPMINER algorithm. Thus, all critical pairs stemming from two rules appearing in P' are joinable.

3) Let (S_1, S_2) be a critical pair stemming from one rule $R_1 \in P$ and one rule $R_2 \in P'$. We distinguish three cases:

   a) R_1 and R_2 are propagation rules. This situation is analogous to case 1.

   b) R_2 is obtained by transforming the propagation rule R_1 into a simplification rule. The heads of R_1 and R_2 are the same. Let S_i be the state resulting from the application of R_i to the critical ancestor state of R_1 and R_2 $(i = 1, 2)$, then applying R_2 to S_1 leads to a variant of S_2. Therefore, the critical pair is joinable.

   c) R_2 is a simplification rule but it is not obtained by transforming the propagation rule R_1 and R_1 is in P'. This situation is analogous to case 2.

   d) R_2 is a simplification rule but it is not obtained by transforming the propagation rule R_1 and R_1 is not in P', i.e. R_1 is transformed into a simplification rule R_1'.
   Let R_1 be of the form $H_1 \wedge A_1 \Rightarrow B_1$ and R_2 of the form $H_2 \wedge A_2 \Leftrightarrow B_2$. Then R_1' is of the form $H_1 \wedge A_1 \Leftrightarrow B_1 \wedge C$, where C is a subconjunction of $H_1 \wedge A_1$. The critical ancestor state of R_1 and R_2 and of R_1' and R_2 is $(H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2))$.
   Since P' is confluent, the critical pair

   $$(S_1,\ S_2) := (H_2 \wedge (A_1{=}A_2) \wedge B_1 \wedge C,\ \ H_1 \wedge (A_1{=}A_2) \wedge B_2)$$

   stemming from the critical ancestor state of R_1' and R_2 is joinable in P'. Therefore, this critical pair is also joinable in $P \cup P'$.
   The critical state stemming from the critical ancestor state of R_1 and R_2 is

   $$(S_3,\ S_2) := (H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2) \wedge B_1,\ \ H_1 \wedge (A_1{=}A_2) \wedge B_2)$$

   R_1' can be applied to S_3 and leads to S_1. Thus, (S_3, S_2) is joinable in $P \cup P'$.

□

Lemma 4.18
Let P be a set of propagation rules and P' be the transformed program using the SIMPMINER algorithm. If $P \cup P'$ is terminating then for any state S if $S \mapsto_P^* S_1$ and $S \mapsto_{P \cup P'}^* S_2$, where S_1 and S_2 are final states, then $builtIn(S_1)$ and $builtIn(S_2)$ are variants.

Proof:
Since $S \mapsto_P^* S_1$ and $P \subseteq P \cup P'$, we have $S \mapsto_{P \cup P'}^* S_1$ where S_1 is not necessarily a final state.

$P \cup P'$ is terminating then $P \cup P'$ is confluent by Lemma 4.17. We have $S \mapsto_{P \cup P'}^* S_2$ and $S \mapsto_{P \cup P'}^* S_1$, then S_1 and S_2 are joinable, i.e. there is a computation of the form $S_1 \mapsto_{P \cup P'}^* S_2'$, where S_2 and S_2' are variants.

Moreover, we can easily see that the rules in $P \cup P'$ can only remove user-defined constraints from S_1 and the added built-in constraints are constraints that have been already added using the propagation rules appearing in P. Thus, we have that $builtIn(S_1)$ and $builtIn(S_2')$ are variants. Therefore, $builtIn(S_1)$ and $builtIn(S_2)$ are also variants.

□

Lemma 4.19

Let P be a set of propagation rules and P' be the transformed program using the SimpMiner algorithm. If $P \cup P'$ is terminating then for any state S, we have if $S \mapsto^*_{P \cup P'} S_1$ and $S \mapsto^*_{P'} S_2$, where S_1 and S_2 are final states, then S_1 and S_2 are variants.

Proof:

Since $S \mapsto^*_{P'} S_2$ and $P' \subseteq P \cup P'$, we have $S \mapsto^*_{P \cup P'} S_2$.

We prove the claim by contradiction. We assume that S_2 is not a final state for $P \cup P'$. Since S_2 is a final state for P', there exists a propagation rule R in P that can be used to perform a computation step on S_2. If we consider the rule R' in P' obtained from R using the SimpMiner algorithm, since R' and R have the same head, then R' can also be used to perform a computation step on S_2, and thus S_2 cannot be a final state for P', which contradicts the hypothesis.

So, S_2 must be a final state for $P \cup P'$. Moreover, since $P \cup P'$ is terminating then $P \cup P'$ is confluent by Lemma 4.17, and thus we have S_1 and S_2 are variants.

□

4.3 Application

In this section, we show first that when we use the propagation rules generated by the PropMiner algorithm in a CLP approach we benefit of a significant search space reduction but also pay a significant overhead due to the rule triggering process. In many cases this overhead leads to an important increase of the execution time even though the search space is greatly reduced.

Then, we show that the transformation method can be used to obtain the same search space reduction but with less overhead. Thus, the experiments show a significant reduction of the execution time.

The application we consider is in the field of digital circuit design: automatic test-pattern generation. Test generation is the process of defining the tests to apply to a circuit in order to detect faults. Among the possible faults in a circuit composed of boolean gates, a very important type of faults is the *stuck-at faults*. A stuck-at fault occurs when the output value of a gate remains constant, i.e. the output value does not change while the input values are modified.

If we consider a gate in a circuit, we can not access directly the input and the output of the gate to test it. So we must find a way to perform the test using only the input and output pins of the whole circuit. The problem is first to find what signal should be applied on the input of the circuit so that the output of the gate of interest will change (if there is no fault). This is called the *control problem*. Secondly, we must determine how to observe the effect of that change on the output pins of the whole circuit. This is called the *observation problem*.

Several proposals have shown that constraint logic programming allows a simple and declarative formulation of the test generation and leads to an efficient solving process. In the following, we use the constraint logic programming approach for automatic test-pattern generation proposed by Van Hentenryck et al [99]. We shortly present this approach below and then describe our experiments.

4.3.1 Automatic test-pattern generation

In this section, we briefly recall the approach of [99] and refer the reader to the original paper for a detailed description. Van Hentenryck et al defined a specific six-valued logic and provided some rules expressed in the form of so-called demons to carry out the constraint propagation.

Each line in the circuit is associated with a variable constrained to take one of the six possible values. The primary inputs are constrained to be 0 or 1. The four other values, $d, \overline{d}, e$ and $\overline{e}$, are needed to materialize the propagation paths from the output of the gate of interest to the output of the whole circuit (the observation problem). For example, the boolean value 1 at the output of the gate of interest will not be propagated through the circuit as a 1 but as a symbolic value denoted by d to materialize the path from the gate of interest towards the output pins of the whole circuit.

Van Hentenryck et al used rules generated by hand to propagate input and output values of the gates within the circuit. Such a rule is for example: if the input arguments of an *and* gate are d and 1 then the output argument is d. The intuitive meaning of this rule is the following: if the output value of the gate of interest (materialized by d) reaches the input of an *and* gate having a 1 as second input, then the output value of the gate of interest is propagated through this *and* gate.

The triggering of the rules is combined with a systematic labeling in a general *constraint and generate* search, commonly used in constraint logic programming.

4.3.2 Experiments

We consider the problem of finding all possible ways to test each gate in a 4-bit adder. This adder is composed of 21 boolean gates and can be defined using four full adders:

$$
\begin{aligned}
&4BitAdder(X_3, X_2, X_1, X_0, Y_3, Y_2, Y_1, Y_0, O, Z3, Z2, Z1, Z0) \Leftrightarrow \\
&\qquad fulladder(X_0, Y_0, 0, Z0, C0) \land \\
&\qquad fulladder(X_1, Y_1, C0, Z1, C1) \land \\
&\qquad fulladder(X_2, Y_2, C1, Z2, C2) \land \\
&\qquad fulladder(X_3, Y_3, C2, Z3, C3) \land \\
&\qquad xor(C_2, C_3, O).
\end{aligned}
$$

Where $4BitAdder(X_3, X_2, X_1, X_0, Y_3, Y_2, Y_1, Y_0, O, Z_3, Z_2, Z_1, Z_0)$ means that the result of adding the 4-bit binary number (X_3, X_2, X_1, X_0) to the 4-bit binary number (Y_3, Y_2, Y_1, Y_0) is (O, Z_3, Z_2, Z_1, Z_0) and $fulladder(X, Y, Z, S, C)$ simply defines a full adder, i.e. addition of two bits X and Y is performed, where Z is the input carry bit, S is the output bit, and C is the output carry bit.

We present the results of three experiments. Each experiment is performed with a different set of rules. In each experiment, we consider in turn each of the 21 gates in the circuit. For each gate, we find all possible ways to test the gate for stuck-at faults and record two different measures: the size of the search space that has been explored and the execution time. The size of the search space is measured using the number of backtracks made by the labeling predicate (i.e. the number of variable assignments that the program made to find all solutions). The execution time is the CPU time used on a Pentium 3 with 256 MBytes of memory and a 500 MHz processor.

The set of rules used for the first experiment contains only single-headed propagation rules generated by the PropMiner algorithm. This generation has been realized using the truth tables of the operators *and*, *or* and *xor* in the six-valued logic of Van Hentenryck et al, and allowing equalities and disequalities in the body of the rules. Examples of rules are:

$$\begin{aligned}
and(X,1,Z) &\Rightarrow X{=}Z.\\
and(X,0,Z) &\Rightarrow X{\neq}\overline{d} \wedge X{\neq}d \wedge Z = 0.\\
or(X,Y,Y) &\Rightarrow X{\neq}\overline{d} \wedge X{\neq}d.\\
or(X,X,Z) &\Rightarrow X{\neq}\overline{d} \wedge X{\neq}d \wedge X = Z.\\
xor(X,Y,Y) &\Rightarrow X{=}0.\\
xor(d,1,Z) &\Rightarrow Z{=}\overline{d}.
\end{aligned}$$

It should be noticed that this set of rules leads to the same propagations as the rules used by Van Hentenryck et al. Exploiting the symmetry of the ternary operators with respect to the the first and second argument (e.g., $and(X,Y,Z) \Leftrightarrow and(Y,X,Z)$), the number of rules can be reduced to 77 rules. So, it is a hard work to produce this set manually, but such a generation by hand remains possible (as it has been done by Van Hentenryck et al).

In the second experiment, we used also automatically generated propagation rules, but in this experiment we take rules with one or two atoms in the head. Example of rules are

$$\begin{aligned}
and(X,Y,0) \wedge or(Z,Y,X) &\Rightarrow X \neq \overline{d} \wedge X \neq d \wedge X{=}Z \wedge Y{=}0.\\
and(X,Y,Z) \wedge xor(Z,1,Y) &\Rightarrow X{=}0 \wedge Y{=}1 \wedge Z{=}0,.
\end{aligned}$$

Even when exploiting the symmetry of the ternary operators this set consists of 619 rules, and cannot reasonably be generated by hand.

Finally, the third experiment has been made using the rules of the second experiment transformed into simplification rules using SimpMiner. For example, the previous rule has been transformed into

$$\begin{aligned}
and(X,Y,0) \wedge or(Z,Y,X) &\Leftrightarrow X \neq \overline{d} \wedge X \neq d \wedge X{=}Z \wedge Y{=}0.\\
and(X,Y,Z) \wedge xor(Z,1,Y) &\Leftrightarrow X{=}0 \wedge Y{=}1 \wedge Z{=}0.
\end{aligned}$$

The comparison between experiment 1 and experiment 2 is given in Table 4.1. For each gate the table gives the following information: size of search space (number of backtracks) and CPU execution time (in seconds) for the first experiment, size of search space and CPU execution time for the second experiment, variation of the size of the search space from the first to the second experiment (absolute variation, Δ abs., and relative variation in percent, Δ %), and finally variation of the CPU execution time from the first to the second experiment (absolute and relative variations). This table shows that in

gate number	experiment 1		experiment 2		search space		CPU time	
	search space	CPU time	search space	CPU time	Δ abs.	Δ %	Δ abs.	Δ %
1	680	1.76	673	6.55	-7	-1.03	+4.79	+272.16
2	5088	44.86	1728	17.01	-3360	-66.04	-27.85	-62.08
3	620	2.45	417	3.34	-203	-32.74	+0.89	+36.33
4	5088	44.93	1728	14.52	-3360	-66.04	-30.41	-67.68
5	680	1.90	517	2.94	-163	-23.97	+1.04	+54.74
6	1780	8.73	1258	21.12	-522	-29.33	+12.39	+141.92
7	5854	48.40	1909	18.85	-3945	-67.39	-29.55	-61.05
8	1780	10.67	909	11.27	-871	-48.93	+0.60	+5.62
9	12778	133.34	1904	19.56	-10874	-85.10	-113.78	-85.33
10	2740	17.00	1263	17.15	-1477	-53.91	+0.15	+0.88
11	4960	40.00	2670	75.66	-2290	-46.17	+35.66	+89.15
12	7268	48.74	3086	62.09	-4182	-57.54	+13.35	+27.39
13	5900	53.32	1997	39.06	-3903	-66.15	-14.26	-26.74
14	13122	132.43	1944	22.16	-11178	-85.19	-110.27	-83.27
15	8340	77.35	2827	60.44	-5513	-66.10	-16.91	-21.86
16	2926	81.21	514	33.85	-2412	-82.43	-47.36	-58.32
17	7692	63.86	2681	44.99	-5011	-65.15	-18.87	-29.55
18	2822	91.08	687	31.02	-2135	-75.66	-60.06	-65.94
19	7884	124.49	1994	38.58	-5890	-74.71	-85.91	-69.01
20	4988	126.93	965	37.32	-4023	-80.65	-89.61	-70.60
21	11712	172.25	2059	43.25	-9653	-82.42	-129.00	-74.89

Table 4.1: Search space reduction with overhead.

this application, the propagation rules with multiple heads generated automatically (experiment 2) can be used to greatly reduce the size of the search space compared to the search space explored using single-headed propagation rules (experiment 1). Unfortunately, the table shows also that in several cases, we should pay for a very important overhead in terms of execution time to handle these more complex rules.

The comparison between experiment 1 and experiment 3 given in Table 4.2 shows that this overhead can be suppressed if we transform the set of propagation rules to a set of propagation and simplification rules using SimpMiner. Moreover, in nearly all cases, the execution time is reduced by more than 50%. Only a very few overhead remains for gate 1. But, it should be noticed that in this particular case the search space reduction is very low and the execution very brief. So, for this gate, we could not expect to observe a real reduction of the execution time because of the fixed-cost we must pay to handle the set of rules.

4.4 Related Work

- In the pioneering paper [20], K. Apt and E. Monfroy proposed an algo-

gate number	experiment 1 search space	experiment 1 CPU time	experiment 3 search space	experiment 3 CPU time	search space Δ abs.	search space Δ %	CPU time Δ abs.	CPU time Δ %
1	680	1.76	673	1.86	-7	-1.03	+0.10	+5.68
2	5088	44.86	1728	5.90	-3360	-66.04	-38.96	-86.85
3	620	2.45	417	0.96	-203	-32.74	-1.49	-60.82
4	5088	44.93	1728	5.18	-3360	-66.04	-39.75	-88.47
5	680	1.90	517	0.86	-163	-23.97	-1.04	-54.74
6	1780	8.73	1258	5.46	-522	-29.33	-3.27	-37.46
7	5854	48.40	1909	4.71	-3945	-67.39	-43.69	-90.27
8	1780	10.67	909	3.11	-871	-48.93	-7.56	-70.85
9	12778	133.34	1904	7.59	-10874	-85.10	-125.75	-94.31
10	2740	17.00	1263	4.62	-1477	-53.91	-12.38	-72.82
11	4960	40.00	2670	20.80	-2290	-46.17	-19.20	-48.00
12	7268	48.74	3086	17.45	-4182	-57.54	-31.29	-64.20
13	5900	53.32	1997	10.89	-3903	-66.15	-42.43	-79.58
14	13122	132.43	1944	9.03	-11178	-85.19	-123.40	-93.18
15	8340	77.35	2827	17.67	-5513	-66.10	-59.68	-77.16
16	2926	81.21	514	12.66	-2412	-82.43	-68.55	-84.41
17	7692	63.86	2681	14.49	-5011	-65.15	-49.37	-77.31
18	2822	91.08	687	14.21	-2135	-75.66	-76.87	-84.40
19	7884	124.49	1994	16.06	-5890	-74.71	-108.43	-87.10
20	4988	126.93	965	15.36	-4023	-80.65	-111.57	-87.90
21	11712	172.25	2059	17.78	-9653	-82.42	-154.47	-89.68

Table 4.2: Search space and execution time reduction.

rithm for generating rules ensuring a weak notion of local consistency, namely *rule consistency*. In contrast to our approach, the user has no possibility to affect the form of the generated rules. Furthermore, only propagation rules can be generated.

Let C be an atomic constraint. The algorithm presented in [20] generates propagation rules of the following form

$$C(X_1,\ldots,X_k) \wedge X_1{=}v_1 \wedge \ldots \wedge X_k{=}v_k \rightarrow Y{\neq}v,$$

where $X_1,\ldots,X_k$ are some variables occurring in C, and $v_1,\ldots v_k, v$ are elements of the domain associated to variables of C, and Y is a variable occurring in C but not in $X_1,\ldots,X_k$.

The algorithm presented in [83] is a combination of the one described in [20] and unification in finite algebra [82]. Similar to [20] the user has here no possibility to specify the form of the rules. The rules generated by this algorithm have the following form:

$$C(X_1,\ldots,X_k) \wedge X_1{=}v_1 \wedge \ldots \wedge X_k{=}v_k \rightarrow B,$$

where now $v_1,\ldots v_k$ are either elements of the domain or free constants to represent symbolically any element of the domain as used in unification in finite algebra [60]. B is a conjunction of equality constraints and membership constraints (e.g. $X \in D$). With the notion of free constants, equality between variables in the right-hand side of rules can be deduced.

For a constraint $c1$[5] defined by the tuples $\{(0,0,1),(1,1,1)\}$ the following propagation rules are generated by the algorithm presented in [20]:

$$\begin{array}{rcl}
c(X_1,X_2,X_3) & \rightarrow & X_3{\neq}0 \\
c(1,X_2,X_3) & \rightarrow & X_2{\neq}0 \\
c(0,X_2,X_3) & \rightarrow & X_2{\neq}1 \\
c(X_1,1,X_3) & \rightarrow & X_1{\neq}0 \\
c(X_1,0,X_3) & \rightarrow & X_1{\neq}1
\end{array}$$

The algorithm presented in [83] generates the following rules for the constraint $c1$:

$$\begin{array}{rcl}
c(X_1,X_2,X_3) & \rightarrow & X_1{\in}\{0,1\} \ \wedge \ X_2{\in}\{0,1\} \ \wedge \ X_3{=}1 \\
c(x_1,X_2,X_3) & \rightarrow & X_2{=}x_1 \ \wedge \ X_3{=}1 \\
c(X_1,X_2,X_3) & \rightarrow & X_1{=}x_2 \ \wedge \ X_3{=}1
\end{array}$$

In contrast to the algorithms presented in [20] and [83] our approach leads to a more compact and more expressive set of rules. For the constraint $c1$, our algorithm PropMiner generates the single propagation rule, if the user specifies that the right hand side of the rules may consist of a conjunction of equality constraints:

$$c1(X_1,X_2,X_3) \ \Rightarrow \ X_1{=}X_2 \ \wedge \ X_3{=}1$$

With the rules generated by the algorithm presented in [20], one propagates from $c1(X_1,X_2,X_3)$ that $X_3{=}1$. With our generated rule we also propagate that $X_1{=}X_2$. This can also be deduced from the rules generated by the algorithm presented in [83]. However, for a constraint $c2$ defined by its tuples $\{(0,1,0),(0,1,1),(1,0,0),(1,1,0)\}$, our algorithm PropMiner can also generate the following rule

$$c2(X_1,X_1,X_3) \ \Rightarrow \ X_1{=}1 \ \wedge \ X_3{=}0$$

Using this rule, one can deduce from $c2(X_1,X_1,X_3)$ that $X_1{=}1 \wedge X_3{=}0$. This cannot be deduced neither by the algorithm presented in [20] nor by the one presented in [83].[6]

[5] The following two examples are taken from [83].

[6] Personal communication with E. Monfroy, Email, March 2000.

Furthermore, in contrast to the algorithms presented in [20] and [83] our algorithm is able to generate rules with a conjunction of constraints in the left hand side of the rules which is an essential feature for non-trivial constraint handling.

- *Generalized Constraint Propagation* [80] extends the propagation mechanism from finite domains to arbitrary domains. The idea is to find a simple *approximation* constraint that is a kind of least upper bound of a set of computed answers to a goal. In contrast our approach where the generation of rules is done once at compile time, generalized propagation is performed at runtime. Furthermore, [80] does not say much about how to compute the approximations and when to perform propagation steps.

- *Inductive Logic Programming* (ILP) is a machine learning technique that has emerged in the beginning of the 90's [75]. In ILP, the user is interested to find out logic programs from examples. In our case, we generate constraint solvers in the form of propagation and simplification rules, using the definition of the constraint predicates. We used techniques also used in ILP (e.g., [79]), and it is important to consider which of the works done in these fields may be used for the generation of constraint solvers.

4.5 Conclusion and Future Work

We have presented a method for generating rule-based constraint solvers for finite constraints given their extensional representation. The generation is performed in two steps. In a first step, propagation rules are generated. This method has been developed based on several techniques used in association rule mining [17] and ILP [75]. In a further step, propagation rules are transformed into simplification rules. The method has been developed based on the confluence notion of Constraint Handling Rules. Compared to the algorithms described in [20] and [83] our approach is able to generate more general and more expressive rules. On one hand, we allow multiple occurrences of variables and conjunction of constraints with shared variables in the left hand side of rules. On the other hand the user has the possibility to specify the form of the right hand side of rules which can consist of more complex constraints than (dis-)equality constraints.

We also gave various examples to show that our approach can be used as a method to derive new constraint solvers.

Up to now, our method is only able to generate rule-based solvers for finite constraints. Currently, we are extending our method to generate solvers for any kind of constraints, e.g. constraints defined intensionally by a constraint logic program. For the maximum predicate defined by the following clauses

$$\begin{array}{lcl} max(A,B,C) & \leftarrow & A \leq B \wedge C{=}B. \\ max(A,B,C) & \leftarrow & B \leq A \wedge C{=}A. \end{array}$$

we would like to generate the following rules

$$\begin{array}{rcl}
max(A,B,C) & \Rightarrow & C \leq B \wedge C \leq A. \\
max(A,A,C) & \Leftrightarrow & C{=}A. \\
max(A,B,C) \wedge C{\neq}B & \Rightarrow & C{=}A. \\
max(A,B,C) \wedge C{\neq}A & \Rightarrow & C{=}B. \\
max(A,B,C) \wedge B \leq A & \Leftrightarrow & C{=}A. \\
max(A,B,C) \wedge A \leq B & \Leftrightarrow & C{=}B.
\end{array}$$

The challenge will be to extend the PropMiner algorithm in such a way that for a given goal a conjunction of constraints has to be extracted approximating all its answers.

Chapter 5

Extension: CHR$^\vee$

It is obvious that there are similarities and differences between CHR computation and SLD resolution.[1] As we have mentioned in the introduction, while some of the differences between CHR and SLD resolution extend the expressive power of CHR w.r.t. SLD resolution (+), others are incompatibilities ($\neq$) or mere technical differences ($\bullet$):

+ While variable bindings are accumulated in a substitution by SLD resolution, a CHR computation uses equality constraints for this purpose. Obviously every substitution can be expressed by equality constraints, but not every built-in constraint can be simulated by substitutions.

• Since CHR does not handle all constraints through rules, but is able to use a built-in constraint solver, there is a distinction between user-defined and built-in constraints.

+ Horn clauses are used by SLD resolution in a way similar to simplification rules in CHR, since the atom A that is unified with the rule head is removed from the goal. However, there is nothing like **propagation rules** for SLD resolution.

+ CHR allows for **multiple atoms in rule heads**.[2] So a rule application may need to match (and, in the case of a simplification rule, consume) several atoms in the user-defined constraint store.

$\neq$ CHR performs **committed choice**, i.e., it does not generate more than one child state from any given state. That is, only a linear computation is performed instead of a traversal of some tree.

$\neq$ A rule head is **matched** rather than unified with some part of the user-defined constraint store. The restriction to matching is needed in order to retain certain completeness properties in the presence of committed choice [68].

[1] A short description of Horn clause programs and SLD resolution can be found in Appendix A.

[2] Multi-headed rules have not only been investigated for CHR, but according to [37] also for variants of logic programming languages, mainly for coordination languages.

+ CHR allows adding **guards** to rules. This is needed also as a consequence of committed choice: Guards must frequently be used to avoid application of inappropriate rules, which cannot be backtracked as in SLD resolution.

In this chapter, we will show how the two incompatibilities can be eliminated by a simple extension of CHR. We allow disjunctions in rule bodies. We call the extended language "CHR$^\vee$" (to pronounce "CHR-or") [14]. In Section 5.1, we give formal operational semantics for CHR$^\vee$. Section 5.2 shows how the extended language can be used as an experimental platform for several declarative paradigms.

5.1 Constraint Handling Rules with Disjunction

5.1.1 Syntax

We use two disjoint sorts of predicate symbols: One sort for *predefined* predicates and one sort for *free* predicates. Intuitively, predefined predicates are defined by means of a logical theory CT in such a way that statements of the form

$$CT \models \forall(C_1 \wedge \cdots \wedge C_n \rightarrow \exists\bar{x}(C_{n+1} \wedge \cdots \wedge C_m)),$$
where $\bar{x}$ is a subset of variables in $C_{n+1}, \dots C_m$

are decidable. The legitimate forms of predefined predicates and their meaning is specified by a constraint domain for some CLP language and the sort of judgments shown above abstracts the behavior of an ideal constraint solver. Free predicates are those *defined* by a CHR$^\vee$ program.

The set of predefined predicates includes $=$ with the usual meaning of syntactic equality.

Like in CHR, we have two basic kinds of rules: *Simplification rules* $H \Leftrightarrow C|B$ and *propagation rules* $H \Rightarrow C|B$, where the *head* H is a non-empty conjunction of free atoms, the *guard* is a conjunction of predefined atoms and the *body* B a goal. A *goal* is a formula constructed from atoms by conjunctions and disjunctions in an arbitrary way; "*true*" denotes the empty conjunction and "*false*" the empty disjunction. A *simple goal* is one without disjunctions. A *CHR$^\vee$ program* is a finite set of simplification and propagation rules.

5.1.2 Operational Semantics

The operational semantics of CHR$^\vee$ is given by a transition system whose states are goals considered as a disjunction of *subgoals*. A subgoal G is *failed* if its built-in constraints (G_{built}) are unsatisfiable and a state is *failed* if all its subgoals are failed.

The computation steps from Figure 2.1 will be used for CHR$^\vee$ programs to subgoals in the same way as they have been used for CHR programs. However, with the extended syntax disjunctions make their way into the state. In order to handle these, we introduce the **Split** computation step (Figure 5.1).

A *computation* for a goal Q in a program P is a sequence $Q = S_0, S_1, \dots$ of states with $S_i \mapsto S_{i+1}$, however so that no step can be applied to a failed subgoal. A *final state* in a computation is either failed or a *successful* one to which no computation step can be applied and which has at least one successful subgoal. Notice that the **Split** step ensures that successful subgoals always are simple.

Split
$(G_1 \vee G_2) \wedge G \mapsto G_1 \wedge G \;\vee\; G_2 \wedge G$

Figure 5.1: Computation step for disjunctions

The **Split** transition can always be applied to a state containing a disjunction. No other condition needs to be satisfied. This transition leads to branching in the computation in the sense that one subgoal is made into two, each of which needs to be processed separately. In Prolog, disjunctions are processed by means of backtracking, one alternative is investigated and the second only if a failure occurs. For CHR$^\vee$, we need to investigate both branches in order to respect the declarative semantics; however, in an implementation this may be done, e.g., by backtracking (storing the results) or by producing copies of parts of the state to be processed in parallel or interleaved.

Spliting implies that a rule with a disjunction in its body is not just syntactic sugar for two clauses without disjunctions, i.e., $H \Leftrightarrow B_1 \vee B_2$ means something different than the combination of $H \Leftrightarrow B_1$ and $H \Leftrightarrow B_2$. In a computation, the use of the rule with the disjunction means that both B_1 and B_2 occur in the subsequent state, whereas using the two other rules means a commitment to one of B_1 and B_2, i.e. one transition is chosen nondeterministically (in the sense of don't-care nondeterminism, i.e., without backtracking).

Example 5.1 Let P be the following CHR$^\vee$ program:

```
p ⇔ q ∨ r.
q ⇔ false.
r ⇔ true.
```

The evaluation of a goal `p` wrt. this program leads to the computation

```
p ↦ q ∨ r  ↦ false ∨ r  ↦ false ∨ true
```

with a successful final state. □

The declarative semantics of CHR$^\vee$ is the same as for CHR. We now present some results relating the operational and declarative semantics of CHR$^\vee$. These results are similar to those for CLP [55, 68].

Theorem 5.2 (Soundness) Let P be a CHR$^\vee$ program and G be a goal and let $\mathcal{P}$ be the logical meaning of P. If G has a computation with answer constraint C then

$$\mathcal{P}, CT \models \forall\,(C \rightarrow G).$$

□

Theorem 5.3 (Completeness) Let P be a CHR$^\vee$ program and G be a goal and let $\mathcal{P}$ be the logical meaning of P. If G has only finite computations and if $\mathcal{P}, CT \models \forall\ (C \rightarrow G)$, then G has computations with answer constraints $C_1, \ldots, C_n$ such that

$$\mathcal{P}, CT \models \forall\ (C \rightarrow C_1 \vee \ldots \vee C_n).$$

□

Theorem 5.3 shows that in CHR$^\vee$ (like in CLP) it is necessary, in general, to combine several computations with answer constraints to establish that $\mathcal{P}, CT \models \forall\ (C \rightarrow G)$ holds. However, in logic programming each declarative answer (logical consequence of the program) is covered by a more general computed answer.

Example 5.4 Let CT be an appropriate constraint theory describing the predefined predicates $\leq$ and $\geq$ as order relations and let P be the following CHR$^\vee$ program

```
p(X,Y) ⇔ X≤Y ∨ X≥Y.
```

Let G be the goal `p(X,Y)`. It is easily verified that $\mathcal{P}, CT \models \forall(\mathit{true} \rightarrow \mathtt{p}(\mathtt{X},\mathtt{Y}))$ but that *true* does not imply either of the answer constraints `X≤Y` and `X≥Y` alone. □

In general, the completeness theorem does not hold, if G has an infinite computation:

Example 5.5 Let P be the following CHR$^\vee$ program:

```
p ⇔ q ∨ r.
q ⇔ q.
```

Let G be `p`. It holds that $\mathcal{P} \models \mathtt{q} \rightarrow \mathtt{p}$. However, G has only one finite computation with final state `r` and $\mathcal{P} \not\models \mathtt{q} \rightarrow \mathtt{r}$. □

5.2 CHR$^\vee$ as an Experimental Platform for Declarative Paradigms

5.2.1 Logic programming with integrity constraints in CHR$^\vee$

In the following, we show how a program P, written in a constraint logic language given by the predefined predicates of CHR$^\vee$, can be written as an equivalent CHR$^\vee$ program $\mathcal{C}P$ and how integrity constraints can be added to $\mathcal{C}P$ [3]. We distinguish predicates into *intensional*, defined by rules, and *extensional* ones, defined by finite sets of ground facts.

For each intensional predicate p defined by a number of clauses in P,

$$p(t_1^1, \ldots, t_n^1) \leftarrow b_1, \ldots, p(t_1^k, \ldots, t_n^k) \leftarrow b_k,$$

$\mathcal{C}P$ has a simplification rule called the *definition rule* for p of the form

$$p(x_1, \ldots, x_n) \Leftrightarrow (x_1 = t_1^1 \wedge \cdots \wedge x_n = t_n^1 \wedge b_1) \vee \cdots \vee (x_1 = t_1^k \wedge \cdots \wedge x_n = t_n^k \wedge b_k);$$

variables x_i do not occur in the original rules for p. For each extensional predicate p defined by a set of facts in P,

$$p(t_1^1, \ldots, t_n^1), \ldots, p(t_1^k, \ldots, t_n^k)$$

$\mathcal{C}P$ has a propagation rule, called the *closing rule for* p, of the form

$$p(x_1, \ldots, x_n) \Rightarrow (x_1 = t_1^1 \wedge \cdots \wedge x_n = t_n^1) \vee \cdots \vee (x_1 = t_1^k \wedge \cdots \wedge x_n = t_n^k)$$

In addition, $\mathcal{C}P$ has one propagation rule called *extensional introduction rule* of the following form, listing all facts of all extensional predicates of P.

$$\top \Rightarrow f_1 \wedge \cdots \wedge f_n$$

Integrity constraints which can be added to $\mathcal{C}P$ are propagation rules of the form

$$e_1 \wedge \cdots \wedge e_n \Rightarrow b$$

where $e_1, \ldots, e_n$ are extensional atoms, b an arbitrary body.

The evaluation of a goal will proceed in a top-down manner, unfolding it via the definition rules for the predicates applied, leading to a state giving sets of "hypotheses" about extensional predicates. The closing rules will prune this set so that only those hypothesis sets that are consistent with the facts of the original logic program are accepted. Notice that closing rules syntactically are special cases of integrity constraints and that they also serve as such, ensuring that no new extensional facts can be added.

Integrity constraints are not necessarily involved in the processing of a goal but there are cases where the integrity constraint, as a kind of semantic optimization, can identify failures without consulting the actual extension (as embedded in closing and extensional introduction rules); this is shown in the example below. The extensional introduction rule ensures that no successful state can be reached without the integrity constraints being checked.

Example 5.6 The following CHR$^\vee$ program defines extensional `father`, `mother`, and `person` predicates and intensional `parent` and `sibling`. The integrity constraints state natural requirements that any set of extensional facts should satisfy. The predicate "$\neq$" is predefined representing syntactic nonequality.

```
% Definition rules
parent(P,C) ⇔ father(P,C) ∨ mother(P,C).
sibling(C1,C2) ⇔ C1≠C2 ∧ parent(P,C1) ∧ parent(P,C2).

% Extensional introduction rule
⊤ ⇒
    father(john,mary) ∧ father(john,peter) ∧
    mother(jane,mary) ∧
    person(john,male) ∧ person(peter,male) ∧
    person(jane,female) ∧ person(mary,female) ∧
    person(paul,male).

% Closing rules
father(X,Y) ⇒ (X=john ∧ Y=mary) ∨ (X=john ∧ Y=peter).
```

```
mother(X,Y) ⇒ (X=jane ∧ Y=mary).
person(X,Y) ⇒
   (X=john ∧ Y=male) ∨ (X=peter ∧ Y=male) ∨
   (X=jane ∧ Y=female) ∨ (X=mary ∧ Y=female) ∨
   (X=paul ∧ Y=male).

% Integrity constraints
father(F1,C) ∧ father(F2,C) ⇒ F1=F2.
mother(M1,C) ∧ mother(M2,C) ⇒ M1=M2.
person(P,G1) ∧ person(P,G2) ⇒ G1=G2.
father(F,C) ⇒ person(F,male) ∧ person(C,S).
mother(M,C) ⇒ person(M,female) ∧ person(C,G).
```

The goal `sibling(peter,mary)` will be unfolded to different subgoals involving `father` and `mother` hypotheses, some of which fail but `father(john,mary)` $\wedge$ `father(john,peter)` survive and the goal succeeds.

When a goal is processed, the representation of extensional predicates by the extensional introduction rule introduces the facts into the state so that the integrity constraints are processed correctly, e.g., the goal $\top$ succeeds, showing that the integrity constraints indeed are satisfied. The goal `sibling(paul,mary)` will need the acceptance of `father(john,paul)` or `mother(jane,paul)` which are rejected by the closing rules and thus the goal fails.

The goal `father(X,Y)` $\wedge$ `mother(X,Y)` can be brought to failure just by checking the integrity constraints.

```
   father(X,Y) ∧ mother(X,Y)
↦ father(X,Y) ∧ mother(X,Y) ∧ person(X,male) ∧ person(Y,S)
↦ father(X,Y) ∧ mother(X,Y) ∧ person(X,male)∧ person(Y,S) ∧
   person(X,female) ∧ person(Y,S1)
↦ father(X,Y) ∧ mother(X,Y) ∧ person(X,male)∧ person(Y,S) ∧
   person(X,female) ∧ person(Y,S1) ∧ male=female
↦ false
```

□

The correctness properties of the transformation described above can be characterized as follows.

- For any positive Horn clause program P, the definition rules and extensional introduction rule of $\mathcal{C}P$ coincide with the Clark completion of P, and the closing rules are logically redundant. Thus the declarative semantics is preserved under the transformation.

  Example 5.7 Consider for example the well-known ternary `append` predicate for lists, which holds if its third argument is a concatenation of the first and the second argument. It is usually implemented by these two Horn clauses:

  ```
  append([],L,L) ← true.
  append([H|L1],L2,[H|L3]) ← append(L1,L2,L3).
  ```

  The corresponding CHR$^\vee$ program consists of the single simplification rule

```
append(X,Y,Z) ⇔
    (    X=[] ∧ Y=L ∧ Z=L
    ∨    X=[H|L1] ∧ Y=L2 ∧ Z=[H|L3] ∧ append(L1, L2, L3) ).
```

According to the declarative semantics of $\text{CHR}^{\vee}$ the logical meaning of this rule is

$$\begin{array}{llll} \forall X,Y,Z\ (& \mathtt{append}(X,Y,Z) \leftrightarrow & & \\ & \exists L,H,L1,L2,L3 & (& X{=}[] \wedge Y{=}L \wedge Z{=}L \\ & & \vee & X{=}[H|L1] \wedge Y{=}L2 \wedge Z{=}[H|L3] \wedge \\ & & & \mathtt{append}(L1,L2,L3))). \end{array}$$

This is trivially equivalent to the completed definition

$$\begin{array}{lll} \forall X,Y,Z\ (& \mathtt{append}(X,Y,Z) \leftrightarrow & \\ & (\quad \exists L\ (X{=}[] \wedge Y{=}L \wedge Z{=}L) & \\ & \vee\quad \exists H,L1,L2,L3\ (& X{=}[H|L1] \wedge Y{=}L2 \wedge Z{=}[H|L3] \wedge \\ & & \mathtt{append}(L1,L2,L3)))). \end{array}$$

of `append` in the original Horn clause program.

□

- A successful subgoal contains a copy of the extensional part of P together with a satisfiable collection of predefined atoms that corresponds to a computed answer: equations characterize a substitution to the variables of the initial goal and if there are other atoms of predefined predicates, they serve as further constraints on those variable.

- If the database does not satisfy its integrity constraints, any initial goal has a failed computation and with the formulation of a suitable computation rule we can get the result that any computation will fail.

- Completeness is obvious for nonrecursive programs, whereas for recursive programs termination is not guaranteed. However, with a suitable computation rule, we can achieve termination analogous to a traditional CLP implementation for the language in which P is written. A completeness result can be formulated which collects the successful subgoals in a perhaps infinite computation.

The formalism allows us also to define new predicates indirectly by means of integrity constraints that cannot be defined in a feasible way in positive constraint logic programs. It is sufficient to illustrate the principle by means of an example.

Example 5.8 We consider the task of extending the program of example 5.6 with a `orphan` predicate with the intended meaning that `orphan(X)` holds for any `X` which is a `person` but has no `father` or `mother`. A definition is required which is valid for any instance of the extensional predicates in the program, which means that an extensional listing of `orphan` facts is unacceptable. This can be expressed by adding the following three rules to the program.

```
orphan(C) ⇒  person(C,G).
orphan(C) ∧ father(F,C) ⇒ false.
orphan(C) ∧ mother(M,C) ⇒ false.
```

The goal `orphan(X)` results in a final state

`(X=john ∧ orphan(john) ∧` E`) ∨ (X=jane ∧ orphan(jane) ∧` E`)`

where E is the conjunction of the extensional facts in the program.

We see that the first rule defines a range for the `orphan`, giving rise to the possible instantiations of `X`, and the two next ones removes those values for `X` that has a `parent`.

This definition cannot be rewritten as a positive definition, and to express it in Prolog, we need to rely on the dubious procedural semantics of Prolog's approximation of negation-as-failure, e.g., as follows. `orphan(X):- person(X,_), \+father(X,_), \+mother(X,_).` □

We return to this example in the next section.

5.2.2 Abduction in CHR$^\vee$

Abductive querying goes beyond what can be formulated by traditional queries concerning membership of the current state of a database: A goal, typically not implied by the database, is stated to the system and the answer describes ways how the goal could be achieved as a consequence of the database by additional hypotheses about the database state.

Abduction is usually defined as the process of reasoning to explanations for a given goal (or observation) according to a general theory that describes the problem domain of the application. The problem is represented by an abductive theory.

An abductive theory is a triple (P, A, IC), where P is a program, A is a set of predicate symbols, called abducibles, which are not defined (or are partially defined) in P, and IC is a set of first order closed formulae, called integrity constraints.

An abductive explanation or solution for a ground goal G is a set M of ground abducible formulae which when added to the program P imply the goal G and satisfy the integrity constraints IC, i.e.

$$P \cup M \models G \text{ and } P \cup M \models IC$$

In general, the initial goal as well as the solution set M may also contain existentially quantified abducibles together with some constraints on these variables [58, 31, 32], and such solutions are also produced by the method we describe below. We ignore the issue of minimality of solutions which often is required for abductive problem, e.g., diagnosis.

We consider here abduction for positive programs of the sort introduced in Section 5.2.1 and with integrity constraints formulated as CHR$^\vee$ rules as described. Abducibles must be chosen among the extensional predicates.

For such an abductive theory (P, A, IC), we define its translation into a CHR$^\vee$ program $\mathcal{C}(P, A, IC)$ similarly to the translation of Section 5.2.1: The only difference is that the closing rule is left out for each abducible predicate; the extensional introduction rule contains as before all extensional facts, including possible initial facts for abducible predicates.

Example 5.9 The indirect definition of the `orphan` predicate considered in example 5.8 is a special case of this translation with `orphan` considered the only abducible predicate. □

We consider a small example also used in [62].

Example 5.10 Consider an abductive framework with the following program and integrity constraint:

```
bird ← albatross.
bird ← penguin.
penguin ∧ flies → false.
```

Predicates `penguin, albatross` and `flies` are the only abducible. Obviously the abductive solutions for `bird ∧ flies` is {`albatross, flies`}. These solutions are obtained by the following CHR$^{\vee}$ program.

```
bird ⇔ albatross ∨ penguin.
penguin ∧ flies ⇒ false.
```

With this program the evaluation of the goal `bird ∧ flies` leads to the following computation:

```
   bird ∧ flies
↦ (albatross ∨ penguin) ∧ flies
↦ (albatross ∧ flies) ∨ (penguin ∧ flies)
↦ (albatross ∧ flies) ∨ false
```

□

In the following, we show that our framework avoids problems with variables that exist in some abduction systems.

Example 5.11 We continue Example 5.6 and let predicates `father` and `mother` (but not `person`) be abducible. The translation of this abductive framework is as shown in example 5.6 with the closing rules for `father` and `mother` removed.

The abductive goal `sibling(paul, mary)` succeeds with a final state which contains two different abductive explanations (i.e., in two different subgoals) `father(john,peter)` and `father(jane,peter)`. The goal `sibling(goofy, mary)` fails since `person` is not abducible, and thus the closing rule for `person` will reject the hypothesis `person(goofy,-)`.

We can illustrate final states including variables by changing the example so that `person` becomes abducible, i.e., we remove the corresponding closing rule.

Now the goal `sibling(goofy, mary)` leads to the following final state where E is the conjunction of extensional facts in the program:

(father(john,goofy) ∧ person(goofy,G) ∧ E)
∨
(mother(jane,goofy) ∧ person(goofy,G) ∧ E)

This first subgoal gives the abductive explanation for the `sibling` observation that `john` is the common `father` of `goofy` and `mary` and that the individual `goofy` must be a `person` whose gender is not necessary to specify. The second subgoal is similar, explaining `sibling` alternatively by means of a `mother fact`.

The goal `sibling(goofy,mickey)` leads to a final state with two successful subgoals, one of which is

```
father(A,goofy) ∧ person(A,male) ∧
person(goofy,B), ∧ father(A,mickey) ∧
person(mickey,C) ∧ E)
```

The present approach to abduction avoids the problems with variables in abducibles that exist in some abduction algorithms. Three persons are necessary in the explanation, `mickey`, `goofy`, and their unknown `father`. The unknown `father` must be `male` and the gender of the the others does not matter.

□

The correctness properties of the translation of abductive frameworks into CHR$^\vee$ programs described above can be summarized as follows; we consider an abductive framework written as a CHR$^\vee$ program $\mathcal{C}(P, A, IC)$ and an initial goal G.

- For any successful subgoal $A \wedge C \wedge E$ in a computation for G, where A are abducible atoms (not necessarily ground), C predefined, and E the extensional facts of P, and any grounding substitution σ which satisfies C, we have that $P \cup A\sigma \models G\sigma$ and $P \cup A\sigma \models IC$.
- If no successful subgoals exists in a computation for G, there are no abductive explanation for (any instance of) G.
- Completeness is obvious for nonrecursive programs; for recursive programs the situation is similar to what we discussed for the translation of nonabductive programs into CHR$^\vee$.

Finally, we notice that this implementation in CHR$^\vee$ of abductive frameworks with integrity constraints is suited for implementing so-called explicit negation [78] so we can claim also to support negation in our framework. The technique is to introduce, for each predicate p, another abducible predicate *not-p* characterized by the integrity constraint $p(\bar{x}) \wedge \textit{not-p}(\bar{x}) \Rightarrow \textit{false}$.

5.2.3 Model Generation with Constraints using CHR$^\vee$

With propagation rules one has a similar behavior as in deductive databases: A fact is not consumed by applying a rule. We can easily show that there is a transformation from deductive databases in which all derivable atoms are ground to propagation rules that compute the same set of atoms. A bottom-up rule of the form $H \leftarrow B$, where the body is non-empty, has to be converted to the propagation rule $B \Rightarrow H$. Since CHR is query-driven, a rule of the form $H \leftarrow \top$ (usually called a *fact*) has to be converted to $\textit{start} \Rightarrow H$. With the goal `?-start` the evaluation of the corresponding CHR program begins, computing its least Herbrand model.

Example 5.12 The following standard ancestor program

```
anc(X,Y) ← par(X,Y).
anc(X,Y) ← anc(X,Z) ∧ par(Z,Y).
par(a,b) ← ⊤.
par(b,c) ← ⊤.
par(c,d) ← ⊤.
```

can be converted to an operationally equivalent CHR program

```
par(X,Y) ⇒ anc(X,Y).
anc(X,Z) ∧ par(Z,Y) ⇒ anc(X,Y).
start ⇒ par(a,b).
start ⇒ par(b,c).
start ⇒ par(c,d).
```

□

With (disjunctive) propagation rules, it is now also possible to write disjunctive logic programs and to evaluate them in a bottom-up manner in the style of Satchmo [69]. Satchmo expects its input to be given in clausal form. In a way similar to the CPUHR-calculus [13], $\mathrm{CHR}^{\vee}$ can handle even disjunctive logic programs with existentially quantified variables and constraints as shown in the following example.

Example 5.13 Consider the following informations that we have about a university and a student: If a student `S` is enrolled in a course `C` at some time `T` and `C` has another course `C'` as a prerequisite, then `S` must also be enrolled in `C'` at some time `T'` before `T`. `john` has only taken courses before `1994` and from `1996` onward. These informations can be formalized by the following propagation rules:

```
enrolled(S, C, T) ∧ prereq(C, C') ⇒ enrolled(S, C', T') ∧ T'<T.
enrolled(john, C, T) ⇒ T<1994 ∨ T ≥1996.
```

The following rules define a part of the constraint solver for "<", and "≥", that is able to handle constraints even with variable arguments. "$<_B$" is a simple predefined predicate that compares two numbers and cannot deal with uninstantiated variables.

```
X<Y ∧ X<Z ⇔ Y<_B Z  |  X<Y.
X<Y ∧ X≥Y ⇔ false.
X≥X  ⇔ true.
```

The first rule means that the constraint `X<Z` (with a given number `Z`) can be omitted if there is another constraint `X<Y` (with a given number `Y`) provided that `Y` is smaller than `Z`. The second rule means that some value `X` cannot be smaller and greater-or-equal to a value `Y` at the same time. The last rule means that a constraint of the form `X≥X` is trivially valid and can be omitted.

Facts are brought into the computation by a $\mathrm{CHR}^{\vee}$ goal. The facts that `john` has taken course `cs100` in `1996` and `cs100` has prerequisite `cs50` lead to the initial state `enrolled(john, cs100, 1996) ∧ prereq(cs100, cs50)`.

The computation is given in Figure 5.2 in a simplified form. The atom `enrolled(john, cs100, 1996)` activates an instance of the second propagation rule and $\mathrm{CHR}^{\vee}$ distinguishes two cases corresponding to the disjuncts on the right hand side of the rule. In the left case, we get a contradiction (`1996<1994`). In the only successful leaf we see that `john` must have taken `cs50` some time before `1994`.

□

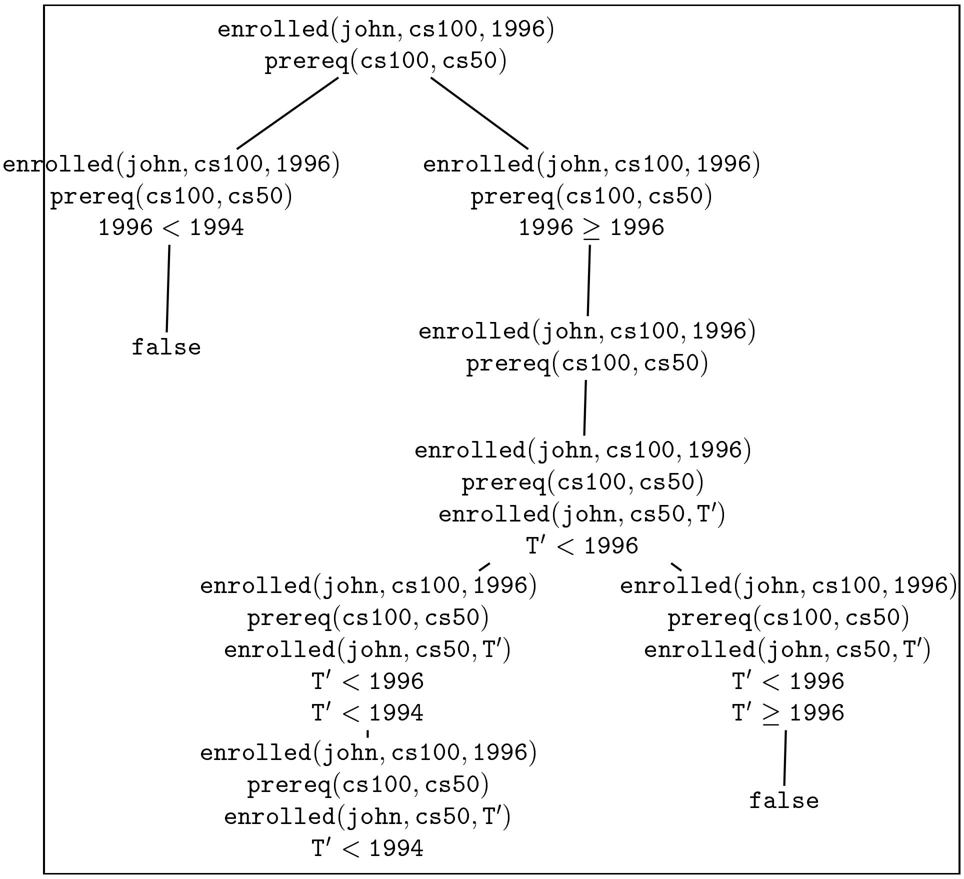


Figure 5.2: Evaluation of Example 5.13

5.2.4 Top-down Evaluation with Tabulation using CHR$^\vee$

In this section, we show how to combine top-down evaluation with Tabulation using CHR$^\vee$.

Consider the following Horn clause program that computes the `N`th Fibonacci number (starting from `0`): `fib(N,M)` is true if `M` is the `N`th Fibonacci number.

```
fib(0,M) ← M = 1.
fib(1,M) ← M = 1.
fib(N,M) ← N≥2 ∧ fib(N-1,M1) ∧ fib(N-2,M2) ∧ M = M1 + M2.
```

The program is extremely inefficient, since Fibonacci numbers are computed several times. The corresponding transformed CHR$^\vee$ program behaves exactly the same as the Horn clause program.

However, with propagation rules one has a similar behavior as in deductive databases: A fact is not consumed by applying a rule. Consider the following CHR$^\vee$ program

```
fib(0,M) ⇒ M = 1.
fib(1,M) ⇒ M = 1.
fib(N,M) ⇒ N≥2 | fib(N-1,M1) ∧ fib(N-2,M2) ∧ M = M1 + M2.
```

and the goal `fib(4,Y)`. The evaluation of `fib(4,Y)` leads to the following final state `Y = 5 ∧ fib(4,5) ∧ fib(3,3) ∧ fib(2,2) ∧ fib(1,1) ∧ fib(0,1) ∧ fib(1,1) ∧ fib(2,2) ∧ fib(1,1) ∧ fib(0,1)`. To improve the efficiency of the program, one has to take into account the computed Fibonacci numbers to avoid that a Fibonacci number will be computed several times.

With the following simplification rule

```
fib(N,M1) ∧ fib(N,M2)  ⇔  M1 = M2 ∧ fib(N,M1).
```

one can reduce the work of the computation by reusing a subgoal which has been evaluated: To compute the 4th Fibonacci number `fib(4,Y)`, the second Fibonacci number `fib(2,Y)` has to be computed twice. This can be avoided with the simplification rule above: The constraints `fib(2,2) ∧ fib(2,Z)` will be reduced to `Z = 2 ∧ fib(2,2)`. In order to achieve the efficiency advantage, the inference engine for $\mathrm{CHR}^{\vee}$ should of course prefer the simplification rule whenever possible.

5.2.5 Combining don't-care and don't-know Nondeterminism

$\mathrm{CHR}^{\vee}$ combines don't-know and don't-care nondeterminism in a declarative way. This can be used for implementing efficient programs as in the following example.

The $\mathrm{CHR}^{\vee}$ program for `append` defined above

```
append(X,Y,Z) ⇔
    (    X=[] ∧ Y=L ∧ Z=L
    ∨    X=[H|L1] ∧ Y=L2 ∧ Z=[H|L3] ∧ append(L1, L2, L3) ).
```

can be improved by adding the following rule

```
append(X,[],Z) ⇔ X = Z.
```

With this rule the recursion over the list X in the initial definition of `append` is replaced by a simple unification $X{=}Z$ if Y is the empty list. In order to achieve the efficiency advantage, we can add a guard of the form `Y` $\neq$ `[]` to the first rule to have non-overlapping rules. This means that in a computation, at most one rule can be chosen for a goal. The second simplification rule corresponds to the admissible guarded rule introduced by Smolka in [94].

5.2.6 Logic Programming and Constraint Solving

In general, writing a constraint application consists of modeling the problem using constraints and then solving them. With $\mathrm{CHR}^{\vee}$ both phases can be done in a uniform language.

We will illustrate the use of our framework for implementing constraint applications by means of the N-queens problem. The problem is to place N queens on a chess board of size $N \times N$ in such a way that no queen can capture another queen. Note that this problem is not hard, in fact one can construct generic solutions for different board sizes, but the program generates all solutions, not just one. Furthermore, the N-queens problem is considered a classical demo for artificial-intelligence programming and it is often used as a benchmark problem to compare different implementations of constraint systems.

There are many different possible models for the N-queens problem. One way of modeling is to recognize that in any solution there is exactly one queen in every column. We associate with the i^{th} column the variable X_i which represents the row number of the queen in that column. In constraint terms, we use N variables with a domain of 1 to N each. Solution are specified as a permutation of the list of the numbers 1 to N. The first element of the list is the row number to place the queen in the first column, the second element indicates the row number to place the queen in the second column, etc. The following program models the N-queens problem using this representation. In the following, we assume that `number` and `=` are predefined predicates. The CHR$^\vee$ programs for `make_domains`, `member` and `remove_values` are omitted for clarity reasons, they can be implemented straightforward.

```
solve(N,Qs) ⇔
    make_domains(N,Qs) ∧ queens(Qs) ∧ labeling(Qs).

queens(L) ⇔
    (    L = []
    ∨    L = [X|Xs] ∧ safe(X,Xs,1) ∧ queens(Xs) ).

safe(X,L,N) ⇔
    (    L = []
    ∨    L = [Y|Rest] ∧ noattack(X,Y,N) ∧ safe(X,Rest,N+1) ).

labeling([]) ⇔  true.
labeling([X|Xs]) ∧ X::L   ⇔  member(X,L) ∧ labeling(Xs).
```

The program works as follow:

- The predicate `make_domains` declares each of the row variables from `Qs` to range over the values `1` to `N`. These correspond to the row variables `X1,...,XN`. To set the variable `X` to have initial domain `L` we use the notation `X::L`.
- The predicate `queens` ensures that no queen falls on the same row or diagonal as any other queen. It iterates through the list of queens calling `safe` to ensure that each queen `X` does not fall on the same row or diagonal as the remaining queens in the list.
- The predicate `safe` iterates through the queens in `Xs` adding `noattack` constraints to enforce that each queen in the list is not on the same row or diagonal as `X`. The implementation of the `noattack` constraint is given below.
- Finally the predicate `labeling` is called, to ensure that a valid solution is found. The predicate `labeling` iterates through each variable `X` in the list of variables to be labeled, calling `member(X,D)`, where `D` is the domain of `X`, to set `X` to each of the remaining values in its domain. The variables are tried in the order of their appearance in the list[3].

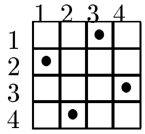
[3]For large N more sophisticated labeling strategies can be implemented.

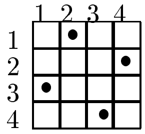
The CHR$^\vee$ program above implements the application-specific part. Now, we want to implement the constraint-solving part. To solve the N-queens problem, our constraint solver only needs to handle the `noattack` constraints. `noattack(X,Y,N)` is true if `Y`$\neq$`X` $\wedge$ `Y`$\neq$`X+N` $\wedge$ `Y`$\neq$`X-N` holds, i.e. `noattack(X,Y,N)` ensures that a queen `Y` is not on the same row or diagonal as a queen `X`.

```
X::[]  ⇔  false.
noattack(X,Y,N) ∧ Y::D <=> number(X) |
                 remove_values([X,X+N,X-N],D,D1) ∧ Y::D1.
noattack(Y,X,N) ∧ Y::D <=> number(X) |
                 remove_values([X,X+N,X-N],D,D1) ∧ Y::D1.
```

The first rule ensures that the domain for `X` cannot be empty. The second and third rule remove the values `X, X+N` and `X-N` from the domain of `Y` provided `X` is a number.

The CHR$^\vee$ program gives two solutions to the goal `solve(4,Qs)`, namely `Qs = [2,4,1,3]` and `Qs = [3,1,4,2]`.





The program executes the goal `solve(4,[X1,X2,X3,X4])` as follows. After the domain declarations the domain is `X1::[1,2,3,4]` $\wedge$ `X2::[1,2,3,4]` $\wedge$ `X3::[1,2,3,4]` $\wedge$ `X4::[1,2,3,4]`. The `safe` predicate adds the following `noattack` constraints: `noattack(X1,X2,1)` $\wedge$ `noattack(X1,X3,2)` $\wedge$ `...` $\wedge$ `noattack(X3,X4,1)`. As each of these constraints involves variables with no fixed value, no propagation occurs. In order to guarantee that a valid solution is found `labeling` is called. The first variable to be assigned is `X1`. Trying the first value in the initial domain, `1`, propagation using the rules from the constraint solving part reduces the domains of `X2`,`X3` and `X4`, i.e. `X2::[3,4]` $\wedge$ `X3::[2,4]` $\wedge$ `X4::[2,3]`. Execution of `labeling` continues until a solution is found.

5.3 Related Work

We briefly compare CHR$^\vee$ to some other (constraint) logic programming languages:

Prolog: We have seen that it is possible to rephrase any pure Prolog program as a CHR$^\vee$ program in such a way that the evaluation of the two programs is equivalent.

CHR$^\vee$ provides a clear distinction between don't-care nondeterminism (committed choice) and don't-know nondeterminism (disjunction), whereas Prolog only supports don't-know nondeterminism in a declarative way. Don't-care nondeterminism is typically implemented in Prolog in a non-declarative way, e.g., using cuts.

MixLog [92] generalizes Prolog in several simple ways:

- Among other things, MixLog provides multiple heads and rules not consuming atoms. This way it supports bottom-up logic programming in a way similar to CHR.
- Furthermore, MixLog can be configured to use committed choice instead of backtracking, but not both in the same program.

In general, it appears that the design of MixLog is mainly driven by the question which features can be supported by simple modifications of the implementation. Thus it looses the close correspondence between the operational semantics and a logic-based declarative semantics, which we think is an important property of logic programming and which is preserved in CHR and CHR$^\vee$.

Sisyphos [102] also adds support for bottom-up logic programming to Prolog. Sisyphos uses separate engines for bottom-up and top-down evaluation. It therefore requires that it is explicitly specified in which direction a rule is to be evaluated. In contrast to this, CHR, CHR$^\vee$ and MixLog allow more gradual transition between rules for the two evaluation directions. The Sisyphos approach, however, has the advantage that it supports calling predicates defined by top-down rules from the antecedents of bottom-up rules.

PROCALOG : There are strong similarities between the work described here and the work of Kowalski et al [62]. Both approaches originated from different starting points but the final result is very similar. Comparing with [62], their proof procedure may involve rewriting of complex formulae that likely can be optimized by methods similar to our use of an underlying CHR environment. Although not investigated in detail, we notice also a similarity between some of our examples and the semantic goal optimizations described in [62, 101].

Oz [93] is a concurrent constraint language for multi-paradigm programming. The user can write customized search strategies, but new constraints can only be defined with a great effort, since multiple heads and propagation rules are missing.

ELAN [59] is an enviroment for specifying and prototyping constraint solvers, theorem provers and deduction systems in general. It also provides a framework for experimenting with their combination. Compared to CHR where one has constraints as a first class concept, the ELAN system is based on labelled conditional rewrite systems and on strategies for controlling their application.

5.4 Conclusion

We have introduced the language CHR$^\vee$, a simple extension of CHR. It supports a new programming style where top-down evaluation, bottom-up evaluation, and constraint solving can be intermixed in a single language. Furthermore, we have shown a straightforward characterization of important aspects of query-answering systems by means of CHR$^\vee$ programs. Programs of constraint logic languages with integrity constraints, important for describing "fine-grained" query evaluation and for constraint databases, can be written directly as $\mathrm{CHR}^{\vee}$ programs. Abductive frameworks fit naturally into this model which provides an implementation that handles correctly a problem with variables in abducibles that exists in some earlier approaches to abduction, e.g., [57, 38]. We can show that indirect characterization of predicates by means of integrity constraints and negation can be expressed also in straightforward ways. This shows that $\mathrm{CHR}^{\vee}$ is useful as a specification language and an implemented, experimental framework for databases and query-answering mechanisms in general. Another important consequence of these results is to demonstrate that efficient implementation techniques for constraint logic programs, as embedded in the underlying CHR environment indeed is applicable for a variety of database applications and query answering mechanisms.

The implementation of $\mathrm{CHR}^{\vee}$ was the most pleasant aspect in our research: Prolog-based implementations of CHR are already able to evaluate $\mathrm{CHR}^{\vee}$ programs. (It happens that even the non-logical features of Prolog such as cut "`!`", `assert/retract` and `ground` work in these implementations in a reasonable way.)

Chapter 6

JACK: A Java Constraint Kit

The constraint programming technology has matured to the point where it is possible to isolate some essential features and offer them as libraries or embedded cleanly in general purpose host programming languages. At the moment, most constraint systems are either extensions of a programming language (often Prolog), e.g. Eclipse, or libraries which are used together with a conventional programming language (often C or C++), e.g. ILOG Solver. Due to the growing popularity of Java and the possibilities of the Internet, there is a big interest to provide constraint handling in Java to implement application servers, e.g. for planning or scheduling systems.

Recently, several proposals have been made to combine the advantages of constraint programming with the advantages of the programming language Java.

- Declarative Java (DJ) [103] provides syntax extensions to Java to support constraint programming. DJ is especially designed to simplify the process of GUI's and Java applets.
- JSolver [34] is a Java library that provides classes to build constraints and strategies to solve these constraints. Thereby it is possible to use variables of the types `int` and `bool`.
- The Java Constraint Library (JCL) [96] provides several algorithms to solve binary constraint satisfaction problems.

In this chapter, we present a new Java library providing constraint programming features. This library is called JACK (JAva Constraint Kit) and consists of three parts:

- JCHR (Java Constraint Handling Rules): A high-level language to write application specific constraint solvers
- VisualCHR: An interactive tool to visualize JCHR computations (See Section 3.3).
- JASE (Java Abstract Search Engine): A generic search engine for JCHR to solve constraint problems

6.1 JCHR: Java Constraint Handling Rules

The first implementations of CHR were interpreters: In 1991 in Eclipse Prolog and in 1993 in Lisp. An interpreter was written in the logical concurrent object oriented language Oz in 1996. In 1994, the first compiler was written as a library of Eclipse [49]. In 1998, an implementation of CHR in Sicstus Prolog [54] was proposed which improves previous implementations in terms of completeness, flexibility and efficency. In this section, we present the first implementation of CHR in Java.

6.1.1 Syntax of a JCHR Solver

A JCHR constraint solver is introduced by the keyword handler followed by the name of the handler and the code of the handler written in curly braces (blocks as known from Java):

```
handler leq {
          ...
            }
```

A JCHR constraint handler consists of three sections: declarations, rules and goals (in that order). Goals for constraints are optional, while a handler without declaring constraints and rules for them would not make much sense. There are two ways of using a constraint handler written in JCHR: Calling it from Java (see Section 6.2) or running it stand-alone using goals. The former will be the usual case in full-fledged applications, while the latter will be helpful for testing and small examples that do not require search (the JASE library, see Section 6.2). When used from Java, the goals of the constraint handler will be ignored. Variables that appear in constraints are called logical variables. Logical variables and class instances must be declared at the beginning of the rules section and at the beginning of each goal in the goals section.

JCHR Declarations

In the declarations section, Java classes are imported and the signatures of the constraints are declared. The Java classes will be needed in the signatures and the code of the rules or goals. The constraints will be implemented in the rules section. As in Java, each declaration is finished by a semicolon. A class import is defined by the keyword class followed by the class name as it can be found in the class path. All classes used in the following need to be imported, including the classes mentioned in the constraint signatures. A constraint is declared by the keyword constraint followed by the name of the constraint and its argument types (much like a Java method):

```
handler leq {
   class java.lang.Integer;
   class IntUtil;
   constraint leq(java.lang.Integer, java.lang.Integer, java.lang.Integer);
```

Rules

In the rules section, first the variables and class instances are declared and then the rules are implemented that simplify the constraints. Variables are defined by the keyword variable followed by a type and variable names:

```
handler leq {
       rules {
            variable java.lang.Integer X, Y, Z;
             ...
             }
            }
```

The rules describe the propagation and simplification of constraints. As in other CHR libraries, there are three kinds of rules: A simplification rule is of the form

```
if Guard { Head } <=> { Body } Name ;
```

A propagation rule is of the form

```
if Guard { Head } ==> { Body } Name ;
```

A simpagation rule is of the form

```
if Guard { Head1 &\& Head2 } <=> { Body } Name ;
```

A rule has an optional name, `Name`, which is a Java identifier. Besides that, a rule consists of an optional guard, a head (left hand side) and a body (right hand side). These parts are all conjunctions using the infix operator `&&`. The head Head is a conjunction of user-defined constraints. The guard is optional. If present the guard is a conjunction of built-in constraints and Java methods. If the guard is not present, it has the same meaning as the guard true. The body Body is a conjunction of user-defined constraints, built-in constraints and Java methods. The built-in constraints are `true` and `false`. Moreover, syntactical equality `==` is provided as a built-in constraint, it can be applied to arbitrary logical variables, regardless of their type.

Goals

Typically, a goal section exists if the constraint solver has to be run stand-alone. If the handler will be used from Java, the goals will be ignored. The goal section consists of one or more goals. Each goal has a name and is introduced by the keyword goal. A goal consists of declarations for the variables and class instances followed by the goal itself. A JCHR goal is a named conjunction of constraints and Java methods (like a rule body).

```
goal g1 {
     variable java.lang.Integer A, B, C;
     leq(A, B) && leq(C, A) && leq(B, C)
        }

goal g2 {
      ...
        }
```

Example 6.1 We will illustrate the syntax of JCHR by the `leq/2` example.

```
handler leq {                                                        (L1)
    class IntUtil;                                                   (L2)
    constraint leq(java.lang.Integer, java.lang.Integer);            (L3)


    rules {                                                          (L4)
        variable java.lang.Integer X, Y, Z;                          (L5)
        { leq(X,X) } <=> { true }                reflexivity;        (L6)
        { leq(X,Y) && leq(Y,X) } <=> { X == Y }    antisymmetry;     (L7)
        { leq(X,Y) && leq(Y,Z) } ==> { leq(X,Z) } transitivity;      (L8)
        { leq(X,Y) &\& leq (X,Y) } <=> { true } idempotence;         (L9)
        if (IntUtil.ground(X) && IntUtil.ground(Y))                  (L10)
             { leq(X,Y) } <=> { IntUtil.le(X,Y) } ground;            (L11)
    }
    goal g1 {                                                        (L12)
        variable java.lang.Integer A, B, C;                          (L13)
        leq(A, B) && leq(C, A) && leq(B, C);                         (L14)
    }
}
```

The first line (`L1`) states that this is the definition of the solver `leq`. In the declaration section, the constraint `leq` is defined by the keyword `constraint` (`L3`). The constraint `leq` expects two arguments of the type `java.lang.Integer`. In the rule section, three variables `X`, `Y` and `Z` of the type `java.lang.Integer` are declared (Line `L4`). They are only used by the rules defined in the rule section. The rule section implements reflexivity, antisymmetry, transitivity, idempotence and a ground rule (`L10-L11`). The reflexivity, antisymmetry and transitivity rules have the same meaning as in Example 3.32. The idempotence rule (`L9`) absorbs multiple occurrences of the same constraint. It can be expressed by a simpagation rule. The last rule states that if the values of `X` and `Y` are known (`L10`) then the constraint `leq(X,Y)` can be replaced by the Java method `IntUtil.le(X,Y)` (`L11`) which is provided by a class `IntUtil` declared in line `L2`. In the goal section, the goal `leq(A,B),leq(C,A),leq(B,C)` is stated.

□

6.1.2 The Prototyping Environment

The JCHR prototyping environment consists of several components. JCHR programs are translated into Java code by the JCHR compiler. It generates Java code which is intended to be integrated into Java applications or applets. The last step which had to be taken to provide CHR for Java was the implementation of an evaluator which is able to interprete the information built with JCHR. It is called the JCHR evaluator. A constraint solver written with JCHR is based on a common constraint system. This system receives information about the used variables, rules, and goals. It is represented in Java by an instance of the class `ConstraintSystem`. This class is also the main part of the evaluator.

A detailed description of the compilation and evaluation scheme and their actual implementation can be found in [89].

JCHR Compiler

With the JCHR compiler, the rules and goals specified in JCHR are translated to Java code. The generated Java code is easy to understand. So it is possible to create rules directly in Java. But the compiler is a useful tool, if CHR is already known from Prolog.

There are three main classes in the compiler package: `JCHRLexer`, `JCHRParser` and `JCHRGenerator`. The class `JCHRLexer` scans a JCHR file and generates a token stream. The class `JCHRParser` generates an abstract syntax tree out of this token stream. The class `JCHRGenerator` transforms this abstract syntax tree into Java code. This segmentation is advantageous concerning modularity. When a modification in JCHR is necessary, it is done in the JCHR grammar file (`jchr.g`). Compiling this grammar file with the ANTLR compiler, the classes `JCHRLexer`, `JCHRParser` and `JCHRParserTokenTypes` are generated [19]. Even the abstract syntax tree (AST) is generated automatically. After this, only the class `JCHRGenerator` which produces the Java classes out of the AST needs to be adapted to the changes.

JCHR Evaluator

The main part of the evaluator is the class `ConstraintSystem`. It provides data structures to store rules and goal constraints and the functionality to evaluate the goal constraints with the rules. In the following, the evaluation algorithm will be explained.

When the constraint system is initialized with rules and a goal, the evaluation process is started with the method `callGoal`. The constraints stored in the constraint system will be evaluated, until the execution of a built-in constraint fails (`false` will be inserted into the built-in constraint memory) or until no rule can be applied to the user-defined constraint memory. This is the case, when all user-defined constraints have been removed from the memory, or when all rules have been tried to be applied on the user-defined constraint memory, the memory state did not change during this period and there is no untested (active) constraint left. If a rule fires for an active constraint, the memories (user defined constraint memory, built-in constraint memory) are updated according to the rule properties. Furthermore, this rule is added to a rule history for debugging purposes. If the currently active constraint has been removed from the user-defined constraint memory (in case of a simplification or a simpagation rule), the next user-defined constraint which is waiting in a queue has to be activated. If there is no untested constraint left, the algorithm finishes. Otherwise, when the currently active constraint has not been removed, the rule loop is reinitialized and the search for firing rules begins again. But now, rules which have already fired with this active constraint may not fire again, at least not with the same combination of partner constraints. When no rule can be found to fire with the currently active constraint, it is set passive and the next constraint in the queue is taken to be active.

Handling of Variables The handling of logical variables in the evaluator is centralized in a variable table in order to keep the occurrences of each variable consistent. Furthermore, the additional information for each variable can be stored in the table. This keeps the variable objects small. A central variable handling presupposes that every constraint used in a rule or as a goal constraint needs to know the variable table. This is done by adding the variable table to each constraint added to the system as a goal constraint and to each rule added to the system as a rule.

Application of Rules The application of a rule consists of

- matching the currently active constraint with a head of a rule
- finding and matching passive partner constraints in constraint store
- checking the guard
- removing matched constraints from constraint store if required
- Executing the body of the rule

The main challenge was to implement the search for finding partner constraints efficiently. We implemented the same approach used in the Sicstus Prolog implementation of CHR [54]. When searching for a partner constraint, a variable common to two constraints in the head of a rule restricts the number of candidate constraints to be checked. Therefore, we *index* the constraint store by the variables shared between partner constraints.

6.1.3 Finite Domain Solver in JCHR

In the following, we present a sample of the finite domain solver implemented in JCHR and the visualization of a goal of the form $X \in \{2,3\} \wedge Y \in \{1,2\} \wedge X \leq Y$ (Figure 6.1).

```
handler fd
{
   class java.lang.Integer;
   class IntUtil;
   class nl; // linked list
   class NlIntUtil;
   class FDUtil;
   class ConstraintSystem;

   constraint fdEnu(java.lang.Integer, nl);
   constraint fdInt(java.lang.Integer,
                    java.lang.Integer,
                    java.lang.Integer);
   constraint fdLe(java.lang.Integer,
                   java.lang.Integer);
   constraint fdLt(java.lang.Integer,
                   java.lang.Integer);
   constraint fdNe(java.lang.Integer,
                   java.lang.Integer);

   rules {
      variable java.lang.Integer X,Y,Z;
      variable java.lang.Integer Min,Max;
      variable java.lang.Integer MinX,MinX1,MinY;
```

```
variable java.lang.Integer MaxY,MaxY1,MaxX;
variable nl L, L1, L2, L3, L4, L5, L6;
variable FDAllDiff AD;

// failure
if (nl.isEmpty(L)) { fdEnu(X, L) } <=>
   { false } failure;

// intersection
{ fdEnu(X, L1) && fdEnu(X, L2) } <=>
   { L = nl.intersection(L1, L2) &&
     fdEnu(X, L) } intersection;

// interaction with intervals
{ fdEnu(X, L) && fdInt(X, Min, Max) } <=>

   { L1 = NlIntUtil.removeLower(Min, L) &&
     L2 = NlIntUtil.removeHigher(Max, L1) &&
     fdEnu(X, L2) } intersection2;

// interaction with inequalities
if (nl.notEmpty(L1) && MinX = NlIntUtil.minList(L1) &&
    nl.notEmpty(L2) && MinY = NlIntUtil.minList(L2) &&
    IntUtil.gt(MinX, MinY))

    { fdLe(X, Y) && fdEnu(X, L1) &&
      fdEnu(Y, L2) } ==>

       { MinX = NlIntUtil.minList(L1) &&
         MaxY = NlIntUtil.maxList(L2) &&
         fdInt(Y, MinX, MaxY) } leMin;

if (nl.notEmpty(L1) && MaxX = NlIntUtil.maxList(L1) &&
    nl.notEmpty(L2) && MaxY = NlIntUtil.maxList(L2) &&
    IntUtil.gt(MaxX, MaxY))

    { fdLe(X, Y) && fdEnu(X, L1) &&
      fdEnu(Y, L2) } ==>

       { MinX = NlIntUtil.minList(L1) &&
         MaxY = NlIntUtil.maxList(L2) &&
         fdInt(X, MinX, MaxY) } leMax;

if (nl.notEmpty(L1) && MinX = NlIntUtil.minList(L1) &&
    nl.notEmpty(L2) && MinY = NlIntUtil.minList(L2) &&
    MinX1 = IntUtil.inc(MinX) &&
    IntUtil.gt(MinX1, MinY))

    { fdLt(X, Y) &&
      fdEnu(X, L1) &&
      fdEnu(Y, L2) } ==>

       { MinX = NlIntUtil.minList(L1) &&
         MinX1 = IntUtil.inc(MinX) &&
```

```
            MaxY = NlIntUtil.maxList(L2) &&
            fdInt(Y, MinX1, MaxY) } ltMin;

    if (nl.notEmpty(L1) && MaxX = NlIntUtil.maxList(L1) &&
        nl.notEmpty(L2) && MaxY = NlIntUtil.maxList(L2) &&
        MaxY1 = IntUtil.dec(MaxY) &&
        IntUtil.lt(MaxY1, MaxX))

        { fdLt(X, Y) &&
          fdEnu(X, L1) &&
          fdEnu(Y, L2) } ==>

           { MinX = NlIntUtil.minList(L1) &&
             MaxY = NlIntUtil.maxList(L2) &&
             MaxY1 = IntUtil.dec(MaxY) &&
             fdInt(X, MinX, MaxY1) } ltMax;

    // interaction with fdNe
    if (NlIntUtil.member(X, L))
        { fdNe(X, Y) && fdEnu(Y, L) } <=>
           { L1 = NlIntUtil.remove(L, X) &&
             fdEnu(Y, L1) } ne1;

    if (NlIntUtil.member(X, L))
        { fdNe(Y, X) && fdEnu(Y, L) } <=>
           { L1 = NlIntUtil.remove(L, X) &&
             fdEnu(Y, L1) } ne2;

    if (NlIntUtil.notMember(X, L))
        { fdEnu(Y, L) &\& fdNe(X, Y) } <=>
           { true } ne3;

    if (NlIntUtil.notMember(X, L))
        { fdEnu(Y, L) &\& fdNe(Y, X) } <=>
           { true } ne4;

    // fdLe, fdLt trivial constraints
    { fdLe(X, Y) && fdLe(Y,X) } <=> { X = Y } leLe;
    { fdLt(X, Y) && fdLt(Y,X) } <=> { false } ltLt;


  }

  goal g1
  {

       variable java.lang.Integer X, Y;

       fdEnu(X,new nl(2,new nl(3))) &&
       fdEnu(Y,new nl(1,new nl(2))) &&
       fdLe(X,Y)
  }

}
```

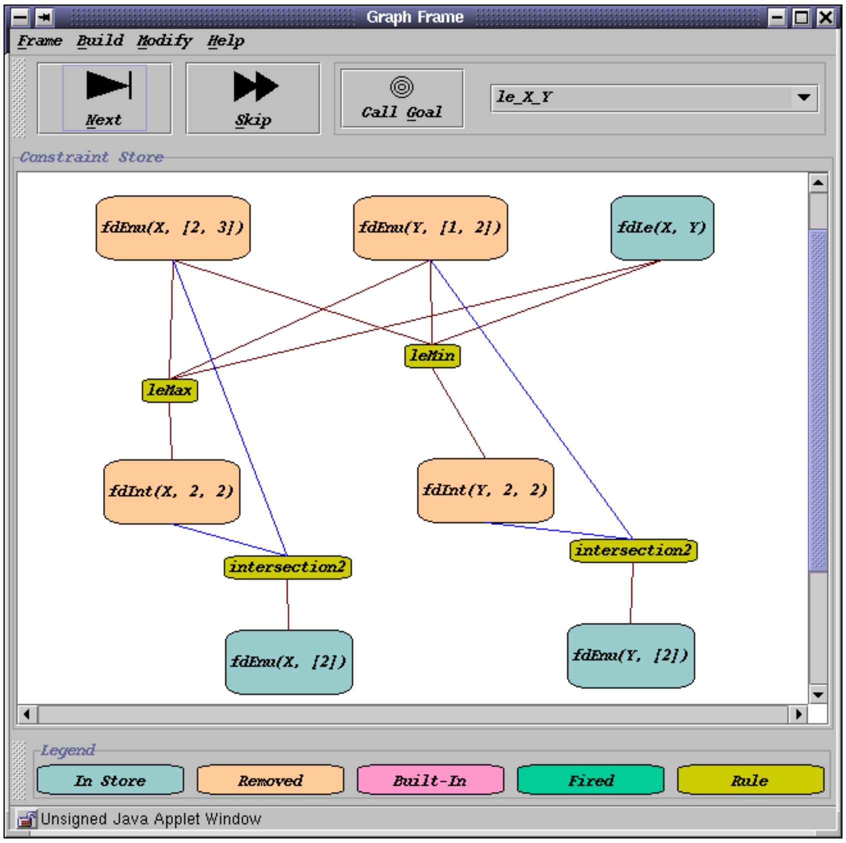


Figure 6.1: Visualization of a goal

6.2 JASE: Java Abstract Search Engine

Usually, constraint reasoning and constraint propagation is incomplete, i.e. it cannot detect inconsistency at all times. Propagation must be combined with search, which is used to assign values to variables. After each assignment step constraint propagation restricts the possible values for the remaining variables, removing inconsistent values or detecting failure. If a failure is detected, the search returns to a previous decision and chooses an alternative. In this section, we present a Java abstract search engine, called JASE[1], which has been actually designed for JCHR but can be used for any Java constraint library. The design of JASE has been inspired by a proposal for the Figaro system [30].

For the search, an *exploration* is used to walk through a *search tree* which consists of *nodes*; the behavior of the nodes is implemented by *choices*.

- A search tree is an unbalanced binary tree. Each inner node of the tree represents one state of the constraint system. Each edge between two nodes represents the transformation of one state into another. Leaf nodes

[1]Pronounced “chase”, because it “chases a solution”

are either empty, and represent a failed constraint system state, or they contain a solution.

- An exploration (interface `SExploration`) implements how to walk through the tree; for example depth-first search.
- A node is implemented by a class called `SNode`. An object of this class stores the constraint system state, and is used by the `SExploration` classes. It does not, however, know about any application details - i.e., what handler is used, which variables are enumerated, which constraints are to be evaluated.
- A choice (point) is implemented by a class that implements the interface `SChoice`. It deals with selecting variables to enumerated, and evaluates constraints. If a new JCHR handler should be enabled for search, (only) this interface must be implemented by a new class.

In the following we illustrate by an example how to use JCHR and JASE.[2] A detailed description of the different classes can be found in [63].

As example for a constraint solver implemented in JCHR, the finite domains solver is used throughout this paper (see Section 6.1.3 for a sample of the finite domain solver). A finite domain constraint is a constraint of the form $X \in D$, where X is a variable, and $D \subset \mathbb{N}$ is a finite subset (domain) of the natural numbers. The textual representation of such a constraint is written as `fdEnu(X,D)`[3] or `fdInt(X,Min,Max)` (which means $X \in D = [Min, Min+1, \ldots, Max]$).

The most important Java object when using JCHR is the `ConstraintSystem`, which encapsulates the constraint solver, the constraint store, and all rules. It is the main object used by the host code.

```
ConstraintSystem cs=new ConstraintSystem();
```

Rules must be inserted in the constraint system; this is done by creating a constraint handler.

```
fdHandler fd=new fdHandler();
fd.defineRules(cs);
```

Constraint variables are represented by Java objects of type `Object`. They are associated with a type and a name.

```
Object X = new Object();
cs.addVariable(X, "java.lang.Integer", "X");
```

The last step in setting up the constraint system is inserting initial constraints with the `addGoalConstraint` method of `ConstraintSystem`. Here, the constraint `fdEnu(X,[2,3,4,5,6])` is created and inserted into the constraint store:

[2] All the code snippets are one contiguous block of source code, they are just separated to insert the explanation text between. Places, where "..." is used, are not relevant for the search, and only contain implementation details.

[3] "`fd`" is the prefix for all finite domain constraints; "`Enu`" stands for "enumeration", which means that `D` is simply a collection of integers.

```
cs.addGoalConstraint(new FDENUConstraint(X,createList(2,6)));
```

Now, the search engine is being set up. In this particular example, the values of the variable `X` should simply be enumerated. So, a container with the variable is created:

```
ObjectContainer vars=new ObjectContainer();
vars.add(X);
```

The next line creates an object that defines what should happen with each solution that is encountered during the search.

```
SChoice collector=new SCollectorChoice(vars,...);
```

The `collector` accumulates all solutions into a container for later use; it is applied to all successful leaves of the search tree ("solutions").

The most important part of the search are the choices made at each node:

```
SChoice rootChoice=new SFDEnuChoice(vars,collector,...);
```

`rootChoice` is the root of the search tree. It is responsible for creating more choices, and it actually modifies and runs the constraint system during the search. The `SFDEnuChoice` used in the example enumerates variables from left to right with no particular heuristic.

Now, the way to explore the search tree is defined (depth-first search).

```
SExploration exploration=
    new SDepthFirstExploration(cs,rootChoice);
```

The search is run, looking for all solutions.

```
boolean success=SSearch.all(exploration);
```

And finally, all solutions can be displayed or otherwise processed.

```
System.out.println(collector.toBeautifulString());
```

Chapter 7

Applications

We present two applications that benefit from using a rule-based language to write constraint solvers. The necessary constraints are expressed and implemented in CHR.

7.1 University Course Timetabling

University course timetabling problems are combinatorial problems which consist in scheduling a set of courses within a given number of rooms and time periods. Solving a real world timetabling problem manually often requires a significant amount of time, sometimes several days or even weeks. Therefore, a lot of research has been invested in order to provide automated support for human timetablers. Contributions come from the fields of operations research (e.g. graph coloring, network flow techniques) and artificial intelligence (e.g. simulated annealing, tabu search, genetic algorithms, constraint satisfaction) [88]. This work refers to terms and methods from constraint satisfaction [65, 64]. The methods presented were developed using CLP.

Applying classical methods from constraint satisfaction requires to model the problem as a *constraint satisfaction problem* (CSP), i.e. a set of variables (representing the points in time courses must begin, for example), each associated with a domain of values it can take on, and a set of constraints among the variables. Constraints are relations which specify the space of solutions by forbidding combinations of values.

The classical CSP framework is of particular interest because many problems from design, resource allocation and decision support (among others) can be cast as CSPs naturally [56, 100]. However, it is not sufficiently expressive for the application under study here. In particular, it does not allow for a distinction between *hard constraints*, which are mandatory, and *soft constraints*, which should get satisfied but may get violated in case this is unavoidable. This limitation forces to treat soft constraints as if they were hard, which frequently leads to over-constrained CSPs without solutions.

Several CSP based frameworks have been introduced which facilitate the formal treatment of soft constraints. For example, hierarchical constraint logic programming [26] allows for constraint hierarchies (a constraint on some level is more important than any set of constraints from lower levels but constraints of the same level are equally important) while in partial constraint satisfaction [44] each constraint is associated with the cost of its violation; see [25] for a more powerful framework, which subsumes other frameworks.

In practice, most constraint-based timetabling systems either do not support soft constraints [22] or use a branch & bound search instead of chronological backtracking [53, 43]. Branch & bound starts out from a solution and requires the next solution to be better. Quality is measured by a suitable cost function that depends on the set of violated soft constraints. With this approach, however, soft constraints play no role in selecting variables and values, i.e. they do not guide search.

Another approach is to adopt techniques developed to propagate hard constraints; soft constraint propagation is intended to associate values with an estimate of how selecting a value will influence solution quality, i.e. which value is known (or expected to) violate soft constraints, or the other way round, which value is known (or expected to) satisfy soft constraints. By considering estimates in value selection, one hopes that the first solution will satisfy a lot of soft constraints. For example, [73] presents a commercial C++ library providing black-box constraint solvers and search methods for the nurse scheduling problem.

Since the black-box approach makes it hard to modify a solver or build a solver over a new domain, our aim was to implement a solver for our timetabling problem using CHR. Inspired by an existing finite domain solver written in CHR we developed a solver which performs hard and soft constraint propagation. The core of the solver takes no more than 20 lines of code. Furthermore, our system, called IfIPlan[1], brought down the time necessary for creating a timetable from a few days by hand to a few minutes on a computer [8, 9].

In the following, we describe the main features of the constraint solver that was used to generate a timetable for the Computer Science Department of the University of Munich. Section 7.1.1 introduces our timetabling problem and the constraints that a solution of the problem had to satisfy. Section 7.1.2 shows how the problem can be modelled as a partial constraint satisfaction problem. Section 7.1.3 gives an overview of the implementation.

7.1.1 Problem Description

The Process of Timetabling

The Computer Science Department at the University of Munich offers a five year program for a master degree in computer science consisting of undergraduate studies (two years) and graduate studies (three years). The problem of timetabling is to be solved every term on base of the timetable of the previous year, the teachers' personal preferences and a given set of courses, each associated with its teachers. The overall process of manual timetabling runs as follows.

After collecting wishes of teachers and information on new courses, a first proposal is developed with the timetable of the previous year as a starting point. This is done by using free slots in the timetable left by courses not taking place again for new courses offered by the same people, whereas wishes of teachers

[1] IfIPlan is an acronym for the German ,,Planer für das Institut für Informatik".

take precedence over the timetable of the previous year. After handing out the proposal to all teachers, evaluations and new wishes are collected.

With the current proposal as a starting point, a next proposal is developed incorporating the responses on the current proposal, again changing as little as possible, and so on. Creating a new timetable is thus a multi-stage, incremental process. Relying on the timetable of the previous year and changing as little as possible by incremental scheduling drastically reduces the amount of work necessary for creating a new timetable and ensures acceptance of the new timetable by keeping the weekly course of events people are accustomed to.

Note that the assignment of rooms is done elsewhere. Nevertheless conflicting requirements for space or certain equipment may be a cause for changing the timetable.

Constraints

The general constraints are due to physical laws, academic reasons and personal preferences of teachers:

- A teacher cannot be at two places the same time, so avoid clashing the courses of a teacher. There should be at least a one hour break between two courses of a teacher.

- Some teachers prefer certain times or days for teaching.

- Monday afternoon is reserved for professors' meetings: Do not schedule professors' courses for Monday afternoon.

- The department consists of five units, each dedicated to a certain area of research. Most courses are held by members of a single unit while only a few courses are held by members of different units. Courses held by members of a certain unit must not clash with courses held by other members of the same unit.

- An offering typically consists of two lectures and a tutorial per week. There should be a day break between the lectures of an offering. The tutorial should not take place on a day, on which a lecture of the same offering takes place. All courses should be scheduled between 9am and 6pm. No lectures should be scheduled for Friday afternoon. No tutorials should be scheduled for late Friday afternoon.

- Only few of the courses are mandatory for and dedicated to students of a certain term while most courses are optional and open to all students. For each term of the undergraduate studies there is a set of mandatory courses, the attendance of which is highly recommended. Courses of the graduate studies only rely on the knowledge provided by courses of the undergraduate studies. There is no recommended order of attendance. Undergraduate courses of a term must not clash, while undergraduate courses of different terms are allowed to clash. Graduate courses should not clash.

Observations and Problems

First observations made clear that existing timetables do not meet the requirements stated, e.g. courses of a unit or graduate courses clash or a lecture of an offering and a tutorial of the same offering are scheduled for the same day. Furthermore, considering the number of graduate courses offered over the years, it became clear that there is too little space to schedule all graduate courses without clashes. This is due to the following reason. As mentioned before, undergraduate courses are mandatory and there is a recommended order of attendance. This way it is possible to distinguish students of the first term from students of the third term and students of the second term from students of the fourth term, which makes it possible to allow clashing of undergraduate courses of different terms. The graduate courses only rely on the knowledge provided by the undergraduate courses. There is no recommended order of attendance thus making it impossible to distinguish students of the fifth term from, e.g., students of the seventh term, which makes it necessary to disallow clashing of graduate courses in some way. So we faced two problems:

- The demand for incremental scheduling by basing the new timetable on the timetable of the previous year and changing as little as possible made it necessary to handle old timetables, which do not meet the requirements stated.
- From a scheduler's point of view the graduate studies lack structure taking freedom and leading to over-constrained timetable specifications.

Tackling the second problem by removing selected no-clash constraints turned out to be laborious and time-consuming and therefore impractical. Classifying graduate courses by contents and expected number of students and allowing clashing of courses of different categories won back some freedom, but it was not possible to identify enough categories in such a way that courses spread evenly over categories, which would have been necessary to prevent conflicts. It became clear that we were in need of some kind of weighted constraints able to express weak and strong constraints that are not mandatory.

7.1.2 A Constraint Model for the Timetabling Problem

A *constraint satisfaction problem* (CSP) [66] (V, C) is a pair, where V is finite set of variables, each associated with a finite domain, and C is a finite set of constraints on these variables. A *solution* of a CSP maps each variable to a value of its domain such that all the constraints are satisfied. Since we have to address quality of a room plan and therefore, have to take into account wishes as well as exploitation of resources, CSP can not model our problem completely. Therefore, we use an extension of the CSP concept. A *partial constraint satisfaction problem* PCSP [44] is a triple (V, C, ω), where (V, C) is a CSP and ω is a total function $\omega : C \to \mathbb{R}$, i.e., ω maps constraints to weights. The weight of a constraint expresses its importance. Thus, one can describe *hard constraints*, which must be satisfied, as well as *soft constraints*, which should be satisfied. A hard constraint is given an infinite weight. Then, a *solution of the PCSP* is an assignment of the variables in V to their domains, such that the total weight of all violated constraints $c \in C$ is minimized.

Clearly, we only need one variable for each course holding the period, i.e. the starting time point, it has been scheduled for. Each variable's domain consists of the whole week, the periods being numbered from 0 to 167, e.g. 9 denotes 9am on Monday, and so on. Requirements, wishes and recommendations can be expressed with a small set of specialized constraints.

- *No-clash constraints* demand that a course must not clash with another one.
- *Preassignment constraints* and *availability constraints* are used to express teachers' preferences and that a course must (not) take place at a certain time.
- *Distribution constraints* make sure that there is at least one day (hour) between a course and another one or that two courses are scheduled for different days.
- *Compactness constraints* make sure that one course will be scheduled directly after another one.

With respect to soft constraints, we chose to distinguish three grades of preferences: weakly preferred, preferred and strongly preferred, which get translated to the integer weights 1, 3 and 9.

7.1.3 Solving the Problem using CHR

Domains

Constraint solving for finite domains constraints is based on consistency techniques [65, 64]. For example, the constraints `X :: [2, 3, 4]`, i.e. `X` must take a value from the list `[2, 3, 4]`, and `X :: [3, 4, 5]` may be replaced by the new constraint `X :: [3, 4]`. Implementing this technique with CHR is straightforward [48]. The first rule ensures that the domain for `X` is non-empty, the second rule intersects two domains for the same variable:

```
X::[]  ⇔  false.
X::L1 ∧ X::L2  ⇔  intersection(L1,L2,L) ∧ X::L.
```

However, this scheme is not sufficient for our needs: Since soft constraints may be violated, the values to be constrained must not be removed from the domain of the variable. Moreover, when we have to choose a value for the variable during search, we must be able to decide whether a certain value is a good choice or not. Therefore, each value must be associated with an assessment. We chose to represent a domain as a list of value-assessment pairs. For example, assume the domain of `X` is `[(3, 0), (4, 1), (5, -1)]`. Then `X` may take one of values `3`, `4` and `5`, whereas `4` is encouraged with assessment `1` and `5` is discouraged with assessment `-1`.

Low-level Constraints

The solver is based on three types of constraints.

- `domain(X, D)` means that `X` must get assigned a value occurring in the list of value-assessment pairs `D`.

- `in(X, L, W)`: Its meaning depends on the weight `W`. If `W = inf`, i.e. if the constraint is hard, it means that `X` must get assigned a value occurring in the list `L`. If `W` is a number, i.e. if the constraint is soft, it means that the assessment for the values occurring in `L` should be increased by `W`.
- `notin(X, L, W)`, if hard, means that `X` must not get assigned any of the values occurring in the list `L`. If it is soft, it means that the assessment for the values occurring in `L` should be decreased by `W`.

The Core of the Solver

Propagating a soft constraint is intended to modify the assessment of the values to be constrained. For example, assume the domain of `X` is `[(3, 0), (4, 1), (5, -1)]` and assume the existence of the constraint `in(X, [3], 2)` stating that `3` should be assigned to `X` with preference `2`. Then we have to increase the assessment for value `3` in the domain of `X` by adding `2` to the current assessment of `3` obtaining the new domain `[(3, 2), (4, 1), (5, -1)]` for `X`. However, applying a hard constraint will still mean to remove values from the variable's domain. Consequently, an `in` constraint is processed by either pruning the domain or increasing the assessment for the given values.

```
fd_in_hard @ domain(X, D) ∧ in(X, L, W) <=> W = inf |
    domain_intersection(D, L, D1) ∧
    domain(X, D1).

fd_in_soft @ domain(X, D) ∧ in(X, L, W) <=> W ≠ inf |
    increase_assessment(W, L, D, D1) ∧
    domain(X, D1).
```

In case a hard `in` constraint has arrived, rule `fd_in_hard` looks for the corresponding `domain` constraint, which contains the current domain `D`, and replaces both by a new `domain` constraint, which contains the new domain `D1`. The domain `D1` results from intersecting `D` with the list of values `L`. Rule `fd_in_soft` works quite similar except for `D1` results from `D` by increasing the assessments for the values occurring in `L`. Note that the guards exclude each other. Therefore, whichever constraint arrives, only one of the rules will be applicable. The rules for `notin` are similar.

```
fd_notin_hard @ domain(X, D) ∧ notin(X, L, W)  ⇔  W = inf |
    domain_subtraction(D, L, D1) ∧
    domain(X, D1).

fd_notin_soft @ domain(X, D) ∧ notin(X, L, W)  ⇔  W ≠ inf |
    decrease_assessment(W, L, D, D1) ∧
    domain(X, D1).
```

Subtracting weights, which are always positive, may result in negative assessments.

Whenever a domain of a variable has been reduced to the empty list, the variable cannot get assigned a value without violating hard constraints. This case is dealt with by the following simplification rule.

```
fd_empty @ domain(_, []) ⇔ false.
```

With only one value left in a domain of a variable we can assign the remaining value to the variable immediately.

```
fd_singleton @ domain(X, [(A, _)]) ⇒ X = A.
```

We use a propagation rule instead of a simplification rule because the `domain` constraint must not be removed. Without it the processing of `in` and `notin` constraints imposed on the domain of a variable would not be guaranteed and thus an inconsistency might be overlooked.

Treatment of Global Constraints

Up to now we only dealt with the low-level constraints of our finite domain solver. Now we exemplify how to express global (n-ary) application-level constraints in terms of `in` and `notin` constraints.

`no_clash(W, Xs)` means that, depending on the weight `W`, the variables from `Xs` must or should get assigned distinct values. It gets translated to `notin` constraints. This translation is data-driven: whenever one of the variables from `Xs` gets assigned a value, this value gets discouraged or forbidden for the other variables by the following rule.

```
fd_no_clash @ no_clash(W, Xs) ⇔
    Xs ≠ [_] ∧
    select_ground_var(Xs, X, XsRest)
    |
    post_notin_constraints(W, X, XsRest) ∧
    no_clash(W, XsRest).
```

The guard first makes sure that `Xs` contains at least two elements. Then it selects a ground variable `X` from `Xs` remembering the other variables in `XsRest`. With no ground variable in `Xs`, the Prolog predicate `select_ground_var` fails. If the guard holds, `no_clash(W, Xs)` gets replaced by

- `notin` constraints produced by the Prolog predicate `post_notin_constraints`, one for each member of `XsRest`, discouraging or forbidding the value `X` and
- a `no_clash` constraint stating that the variables in `XsRest` should or must get assigned distinct values.

Note that the predicate `post_notin_constraints` fails in case `XsRest` contains the value `X`.

A singleton list of variables means that there is nothing more to do. This case is handled by the following rule.

```
fd_no_clash_singleton @ no_clash(_, [_]) ⇔ true.
```

The translation of the other application-level constraints either follows this scheme or is a one-to-one translation.

Interaction of the `no_clash` Rules and the the Core of the Solver

In the following, we present two examples to show how the CHR rules interact with each other. In the first example, we deal only with hard constraints. Assume the current state of a computation consists of the constraints

```
    domain(X, [(1, 0), (2, 0)]),
    domain(Y, [(1, 0), (2, 0)])
and no_clash(inf, [X, Y]).
```

Since neither `X` nor `Y` are ground, no rule is applicable. After adding the constraint `in(X, [1], inf)` rule `fd_in_hard` becomes applicable and simplifies

```
    domain(X, [(1, 0), (2, 0)])
and in(X, [1], inf)
 to domain(X, [(1, 0)]).
```

Now rule `fd_singleton` becomes applicable and propagates the equality constraint `X = 1`. Then rule `fd_no_clash` becomes applicable and simplifies

```
    no_clash(inf, [1, Y])
 to notin(Y, [1], inf)
and no_clash(inf, [Y]).
```

Then rules `fd_no_clash_singleton` and `fd_notin_hard` become applicable: rule `fd_no_clash_singleton` removes `no_clash(inf, [Y])` and rule `fd_notin_hard` simplifies

```
    domain(Y, [(1, 0), (2, 0)])
and notin(Y, [1], inf)
 to domain(Y, [(2, 0)]).
```

Finally, rule `fd_singleton` becomes applicable and propagates the equality constraint `Y = 2`. Thus, the final state of the computation consists of

```
    domain(X, [(1, 0)]),
    domain(Y, [(2, 0)]),
    X = 1
and Y = 2.
```

In the second example, we want to show how the rules treat soft `no_clash` constraints. Assume the current state of a computation consists of the constraints

```
    domain(X, [(1, 0), (2, 0)]),
    domain(Y, [(1, 0), (2, 0)])
and no_clash(1, [X, Y]).
```

Again we add `in(X, [1], inf)`. Until rule `fd_no_clash` becomes applicable, the computation proceeds as before. Then rule `fd_no_clash` simplifies

```
    no_clash(1, [1, Y])
 to notin(Y, [1], 1)
and no_clash(1, [Y]).
```

Finally, both rules `fd_no_clash_singleton` and `fd_notin_soft` become applicable: rule `fd_no_clash_singleton` removes `no_clash(1, [Y])` and rule `fd_notin_soft` simplifies

```
    domain(Y, [(1, 0), (2, 0)])
and notin(Y, [1], 1)
 to domain(Y, [(1, -1), (2, 0)]).
```

Thus, the final state of the computation consists of

```
    domain(X, [(1, 0)]),
    domain(Y, [(1, -1), (2, 0)])
and X = 1.
```

Propagation Performance

The first rule for `no_clash` (`fd_no_clash`) acts as a constraint propagator that amplifies the constraint store by incrementally spanning a network of `notin` constraints. Since the propagator sleeps as long as none of the variables it surveys gets assigned a value, a `no_clash` constraint cannot contribute to a solution as long as none of its courses gets scheduled. This approach is similar to the implementation of CHIP's `all_different` constraint [97].

Combining our solver with chronological backtracking results in a search procedure, which, with respect to propagation performance, is a little better than the forward checking algorithm [52] and much worse than the generalized arc-consistency algorithm [74].

Concerning the reuseability of our solver, we cannot give a definite answer. On the one hand, experience shows that, for a variety of problems, forward checking together with additional search is more efficient than applying more expensive consistency techniques [64]. On the other hand, there is evidence that maintaining arc-consistency is necessary to solve the larger and the harder problems efficiently [24, 85, 71].

Whether the performance of our solver is sufficient to solve a whole university timetabling problem depends on the structure of the problem. If departments share teachers, students, rooms or equipment and sharing has to be taken into account, the problem might be too hard. Otherwise, the university timetabling problem breaks down into several independent timetabling problems, one for each department. This is the case with most German universities.

The Search Procedure

The search procedure employed integrates the solver given above with chronological backtracking and heuristics for variable and value selection. For variable selection, we chose the first fail principle [52] which dynamically orders variables by increasing cardinality of domains, i.e. the principle proposes to select one of the variables with the smallest domains with respect to the current state of computation. For value selection, we used a best-fit strategy choosing one of the best-rated periods. From an optimistic point of view, this will be one of the periods violating a set of soft constraints with minimal total weight, but the estimate may be too good due to the low propagation performance of the `no_clash` solver. Furthermore, the best assessment does not necessarily violate a minimum number of constraints: a strong personal preference may balance out ten weak no-clash constraints. This approach yielded a good first solution to our problem. It was not necessary to search for a better solution.

Generation of Timetables

The generation of a timetable runs as follows. Each course is associated with a `domain` constraint allowing for the whole week, the periods being numbered from 0 to 167. It is important to note that, for each course, the initial assessment for all periods is 0 indicating that no period is given preference initially. Then preassignment constraints and availability constraints will be translated into `in` and `notin` constraints. Adding `in` and `notin` constraints may narrow the domains of the courses using the rules presented above. Propagation continues until a fixpoint is reached, that is to say, when further rewriting does not change the store. Usually, our consistency based finite domain solver is not powerful enough to determine that the constraints are satisfiable. In order to guarantee that a valid solution is found the search procedure is called. Addition of an `in` constraint may initiate propagation, and so on.

Now, that we have discussed the details of creating a timetable, how do we create a new timetable based on a timetable of the previous year with our system? Central to our solution is the notion of *fixing a timetable*. Fixing a timetable consists in adding a (strongly preferred) soft preassignment constraint for each course that has been scheduled ensuring that all courses offered again will be scheduled for the same time.

The time necessary to compute a timetable depends on whether a previous timetable is reused or not. Scheduling 89 courses within 42 time periods from scratch took about five minutes. Considering an "almost good" previous timetable saved about two and a half minutes.

7.2 Classroom Assignment

In most universities, the university course timetabling problem is solved in two phases. In the first phase timetables have to be created, one for each department. Since departments can share rooms, the availability of rooms is not taken into account in the first phase. In the second phase, rooms have to be assigned to courses. The assignment of rooms is done centrally for the whole university.

The classroom assignment problem is a difficult and time-consuming expert task since a lot of requirements have to be met. For example, courses must be assigned to rooms based on the number of students taking the courses and capacities of rooms. Furthermore, some courses may require special equipments such as beamer or internet access. While for the first phase of the university course timetabling several systems have been developed [22, 53, 43], to our knowledge mainly theoretical work has been done on the topic of the classroom assignment problem [42, 28].

In our approach, the generation of classroom plans for universities is tackled using the CLP framework. The system is called *RoomPlan* and is currently tested at the University of Munich [12]. Our prototype brought down the time necessary for creating a classroom plan from a few days by hand to a few minutes on a computer.

Usually not all specified requirements can be fulfilled since the number of (special) rooms is obviously limited. We distinguish between hard and soft constraints. The classical approach to deal with these requirements is based on a variant of branch & bound search. Usually, the computation of the cost function is incorporated into the labeling process. In the following, we propose another approach computing the cost function during the constraint solving process independent of the labeling procedure. This requires to modify the constraint solving part.

For our need, we extended an existing finite domain solver written in CHR in a way that the cost for a solution is computed during the propagation of soft constraints. CHR allows to express the calculation of the cost in a very declarative and straightforward manner.

In the following, we describe the main features of the constraint solver that was used to generate a classroom plan for the University of Munich. Section 7.2.1 introduces our classroom assignment problem and the constraints that a solution of the problem had to satisfy. Section 7.2.2 shows how the problem can be modelled as a partial constraint satisfaction problem. Section 7.2.3 gives an overview of the implementation.

7.2.1 Problem Description

In universities, where each department is responsible for its own timetable and where rooms can be shared by different departments, timetables are usually generated in two phases. In the first phase an assignment of courses within a given number of periods is done without taking into account the availability of rooms. This task has to be performed separately for each department. For the Computer Science Department at the University of Munich, the first phase is solved automatically by a system that generates a new timetable based on a timetable of the previous year (See Section 7.1). For other departments the generation of timetables is still done by hand. In the second phase an assignment of courses within a given number of rooms has to be performed. After collecting timetables of all departments and wishes of teachers a classroom plan is generated centrally.

In the following, we want to investigate the classroom assignment problem of the University of Munich. Since timetables for departments change every semester, a new classroom plan has to be created each semester. The University of Munich dispose of different buildings. The biggest building consists of 40 rooms where about 1000 courses have to be held.

The generation of a classroom plan is a difficult and time-consuming task since different kinds of constraints have to be taken into account:

- The *no-occupation overlap constraint* tells that occupation time of a room by courses must not overlap.
- The *seat requirement constraint* tells how many seats a course requires.
- The *teacher's wishes*: We distinguish three kinds of wishes.
  - A *room constraint* binds a course date to a room.
  - A *building constraint* assigns a course date to a certain building.
  - An *equipment constraint* constrains a course date to be assigned to a room with certain technical equipment, e.g. beamer or video.

Usually not all specified requirements can be fulfilled since the number of (special) rooms is obviously limited. Therefore we distinguish between hard and soft constraints. Roughly speaking, *no-occupation overlap constraints* and *seat requirement constraints* determine hard constraints, wishes may be hard or soft constraints.

7.2.2 A Constraint Model for Classroom Assignment

Now, the classroom assignment problem is modeled as a PCSP. Note, that it does not suffice to assign a room to each course, but instead we have to assign a room to each date, when a course is held. Therefore, we use one variable for each *course date*. For example, if a course consists of two lectures, the course is represented by two different course date variables. The initial domain of each course date variable is the set of all rooms in the university. Thus, the solution is an assignment of course dates to rooms.

There are two constraints that occur only as hard constraints and thus have infinite weight: the *no-occupation overlap constraint* and the *seat requirement constraint*. Wishes may also be hard, i.e. have infinite weight.

To ensure a good exploitation of resources by a solution, we evaluate assignments of a room to a course date. For this reason, we modify the weight $\omega(\alpha)$ for the constraint α, that assigns a course date for a course c to a room r. This is done by adding a term to the user-defined evaluation $\omega'(\alpha)$, thus defining $\omega(\alpha)$ in the following way.

$$\begin{aligned} \omega(\alpha) &= \omega'(\alpha) + a_1 \cdot \frac{\text{seats}_r - \text{requirement}_c^{\text{seat}}}{\text{seats}_r} + \\ &\quad \min(0, a_2 \cdot \frac{\text{equipment}_r - \text{requirement}_c^{\text{equipment}}}{\text{equipment}_r}), \end{aligned} \tag{7.1}$$

where seats_r is the number of seats in room r, $\text{requirement}_c^{\text{seat}}$ is the number of seats required by course c, equipment_r is a valuation of the technical equipment in r and $\text{requirement}_c^{\text{equipment}}$ a value for the technical requirements of c. a_1 and a_2 are negative constants weighting the exploitation of seats and equipment resources against each other and the violation of wishes. Since the *equipment constraint* can be soft, the value of the technical requirements can be greater than the value of the equipment. In this case, we have to avoid that the third term of the function $\omega(\alpha)$ is positive.

7.2.3 Solving the Problem using CHR

In a PCSP, one has additionally to satisfying all hard constraints to take soft constraints into account. According to the PCSP model, we have to minimize the total weight of violated soft constraints. This is equivalent to maximize the total weight of satisfied constraints. We use a *branch & bound* approach to tackle this maximization problem. Branch & bound is a standard method to optimize a score that works by constraining the score during the search. Every time an assignment satisfying the hard constraints is found, the score is bound to be even better. Thus, the last assignment compatible to the hard constraints that is found will have an optimal score. Therefore, we incrementally compute a bound of the score, that an assignment compatible to the current hard constraints may have, during the enumeration. This way we prune the search tree every time the maximally achievable score is worse than the score of the previous solution.

To prune the search tree efficiently in our branch & bound algorithm, we have to keep track of the upper bound of the score. The upper bound of the score may be affected each time the constraint store changes. This change may be done either by a constraint which is directly inserted by the labeling process or by constraint propagation. If only changes of the first kind could affect the upper bound, the calculation of the score could be easily incorporated into the labeling process. However, since we also have to take care of the second kind of constraint store changes, it is much more natural and intuitive to do this calculation concurrently to the labeling process and triggered by the alteration of the constraint store. Constraint Handling Rules (CHR) allows to express this in a very declarative and straightforward manner, where the calculation is formulated independently of the labeling.

Handling Hard Constraints

Regarding just the hard constraints, our solver is essentially a finite domain solver, i.e. a course date variable is bound to a list of rooms and constraints may eliminate rooms from the domain list of the constrained variable.

To solve our classroom assignment problem, the *no-occupation overlap constraint* can be expressed using the global constraint `all_distinct`. The constraint `all_distinct(Xs)` tells that all variables in the list `Xs` must be bound to different values. The *no-occupation overlap constraint* is propagated to constraints `all_distinct(Xs)`, where the `Xs` are lists of the course date variables for all course dates that overlap in time. The hard *seat requirement constraint* propagates by filtering the domains of the course date variables.

Handling Soft Constraints

In the following, the calculation of the score done by a CHR program is described. It is most intuitive to calculate the total score from three sub-scores, which can easily be done by a rule. One sub-score `ScoreWish` is the total weight of satisfied wishes, the second `ScoreSeatRes` is the sum of the second terms in equation (1) over all assignments of course dates to rooms, i.e. a measure of the exploitation of seats. Analogously, the last `ScoreEquipmentRes` is the sum of the third terms in equation (1), i.e. a measure of the exploitation of equipment. The total score is computed as a weighted sum of the sub-scores. Instead of minimizing the total weight of violated constraints, we equivalently maximize the total weight of the satisfied soft constraints. Since we maximize the score we need to compute an upper bound of the score, that an assignment which satisfies the hard constraints of the current constraint store may have.

We start with describing the computation of the sub-score `ScoreWish`. The different types of wishes are expressed by CHR constraints of the form `wish(Type, CourseDate, Wished, Weight)`, where the first argument gives the type of the wish, namely `room`, `building` or `equipment`. `CourseDate` holds a course date identifier, the variable `Wished` specifies the instance of the wish and `Weight` holds the weight of the wish. We use the constraint `assignment(CDate,CDateVar)` which tells that `CDateVar` is the course date variable corresponding to the course date `CDate`. The constraint `scoreWish(Up)` tells that `Up` is the upper bound of the sub-score `ScoreWish`.

In the following, we introduce rules to update the sub-score `ScoreWish`, whenever a wish is satisfied or violated. In the following, we will discuss the rules for handling *room constraints*. The rules handling *building* and *equipment constraints* are analogous.

```
assignment(CDate, CDateVar) ∧ CDateVar::Dom \
wish(room, CDate, RoomWish, infinite) ⇔
       CDateVar::[RoomWish].

assignment(CDate, CDateVar) ∧ CDateVar::[RoomWish] \
wish(room, CDate, RoomWish, Weight) ⇔
                Weight ≠ infinite | true.

assignment(CDate, CDateVar) ∧ CDateVar::Dom  \
wish(room, CDate, RoomWish, Weight) ∧ scoreWish(Up) ⇔
                Weight ≠ infinite ∧ not member(RoomWish, Dom) |
       scoreWish(Up - Weight).
```

The first rule propagates a hard wish, i.e. a wish with infinite weight, in such a way that an assignment of the course date variable to the wished room is performed. Therefore, the simpagation rule tests whether constraints of the form `assignment(CDate, CDateVar)`, `CDateVar::Dom` and `wish(room, CDate, RoomWish, infinite)` can be found in the constraint store. In this case, the rule fires and the constraint `CDateVar::[RoomWish]`, that binds the room to the wished room, is added to the store. Furthermore, the `wish` constraint is removed from the constraint store. The application of this rule leads to the occurrence of two domains for the same variable. These constraints can be simplified by the intersection rule presented above.

The second rule handles already satisfied soft constraints. If the constraints `assignment(CDate, CDateVar)`, `CDateVar::[RoomWish]` and `wish(room, CDate, RoomWish, Weight)` are found in the constraint store, then the wish is obviously satisfied and the rule consequently removes the wish. The guard ensures that only soft constraints, i.e. wishes with finite weight, are handled by this rule, since hard `wish` constraints are already handled by the first rule. Note that the upper bound of the score is unaffected if a wish is satisfied. In contrast, if a wish is violated, the upper bound of the score has to be decreased.

The updating of the bound is done by the third rule. The guard ensures that only soft constraints which are violated lead to a change of the score. A *room constraint* is violated if the wished room is not contained in the domain of the course date variable. If the rule fires, the upper bound is recomputed as `Up - Weight` and the constraint `scoreWish(Up - Weight)` replaces the constraint `scoreWish(Up)`.

The evaluation of resource exploitation is handled by a single rule.

```
assignment(CDate, CDateVar) ∧ CDateVar::[RoomNr] \
scoreSeatsRes(UpS) ∧ scoreEquipmentRes(UpT) ⇔
       scoreSeatsRes_diff(CDate, RoomNr, SSD) ∧
       scoreEquipmentRes_diff(CDate, RoomNr, TSD) ∧
       scoreSeatsRes(UpS + SSD) ∧
       scoreEquipmentRes(UpT + TSD).
```

The rule updates the scores for seat resource and equipment resource exploitation, where the actual weight is calculated by the predicates `scoreSeatsRes_diff` and `scoreEquipmentRes_diff` analogously to equation (7.1). The updating of the sub-scores is done by replacing the constraints `scoreSeatsRes(UpS)` and `scoreEquipmentRes(UpT)` by the recomputed constraints `scoreSeatsRes(UpS`

`+ SSD)` and `scoreEquipmentRes(UpT + TSD)` each time an assignment of a course date variable to a room was newly created, either by labeling or by constraint propagation.

Each time a sub-score is recomputed, the total score has to be recalculated. This can be done by the following propagation rule.

```
scoreWish(UpW) ∧ scoreSeatsRes(UpS) ∧ scoreEquipmentRes(UpT) ⇒
       score(UpC + UpS + UpT).
```

The rule fires if a sub-score changes. Then, a constraint with the newly calculated upper bound of the total score is inserted. A further rule ensures that only the most restrictive `score` constraint remains in the constraint store.

```
score(A) \ score(B)  ⇔  A≤B | true.
```

This rule removes the larger of two upper bounds of the total score.

Now, the upper bound can be used to prune the search tree, since we use a branch & bound algorithm. Every time an assignment of the course date variables that satisfies the hard constraints is found, we insert a constraint `last_score(Score)` with the score of this assignment. The following propagation rule ensures, that only better assignments can be found in consequence.

```
last_score(LastScore) ∧ score(Up)  ⇔  LastScore≥Up | false.
```

This rule causes the constraint solver to fail whenever the score of an assignment in the current branch cannot be better than the last score. This is indicated by the upper bound of the score.

Labeling

After stating the problem constraints, normally there are still many feasible solutions, so it is necessary to label the domain variables, i.e. assign them with values that remain on their domains. For the labeling one needs to apply a heuristic strategy that tends to enumerate high scoring solutions early in the search. In a branch & bound search this helps to prune the search tree. In practice also suboptimal solutions may be appropriate, which also emphasizes the use of finding good solutions early.

We employed a *leftmost* variable, *leftmost* value strategy that selects for each assignment the leftmost course date variable and the leftmost value from the domain of the selected course date variable. Since we sort the list of variables as well as the values in the domains before the labeling as we describe below, this strategy prefers most constrained assignments. The sorting is done in the following way. We compute a weight for each course and a weight for each room with respect to a certain course, i.e. actually a weight for a certain assignment. The weights for courses respect seat and equipment requirements, such that courses with strong requirements get great weights. The weight of an assignment totals the weights of the soft constraints which are satisfied by the actual assignment. Then, the course date variables and the rooms in each domain are sorted in descending order by these weights. As a consequence the leftmost course date variables, which are selected first by our strategy, belong to the courses with the strongest requirements. Further, the leftmost values in each domain lead to assignments, which satisfy the "best" soft constraints.

7.3 Conclusion

In this chapter, we have argued that CHR is a good vehicle for implementing finite domain solvers, which performs hard and soft constraint propagation. These solvers are powerful enough to serve as the core of a university timetabling system and a classroom assignment system.

Both prototypes have been designed to meet the specific requirements of the University of Munich. However, it can be applied to other universities, since adaption can easily be done due to the the declarativity of constraint logic programming. The university timetabling system, IfIPlan has been in use at the Computer Science Department of the University of Munich for six terms. The classroom assignment system, *RoomPlan*, is currently tested at the University of Munich. Typically, for 1000 courses and 40 rooms, *RoomPlan* generates a satisfying schedule within a few minutes.

Chapter 8

Conclusions

The present work contributes to a number of research directions in the area of rule-based constraint programming.

- Programming environments are essential for the acceptance of programming languages. Thus, the first step was to propose static and dynamic analysis methods for the rule-based constraint language CHR (Chapter 3).
- Writing constraint solvers remains a hard task even in a rule-based formalism since the programmer has to determine the propagation algorithms. We have proposed a method to generate automatically the propagation and simplification process of constraints (defined over finite domains) in form of rules (Chapter 4).
- To use CHR as a general-purpose logic language, especially for search-oriented problems, we have introduced the language $\mathrm{CHR}^{\vee}$, a simple extension of CHR, and have shown that $\mathrm{CHR}^{\vee}$ is useful as a specification language and an implemented, experimental framework for databases and query-answering mechanisms in general (Chapter 5).
- We have described the design of a Java Constraint Kit, called JACK, consisting of a high-level language for writing constraint solvers (JCHR), an interactive tool to visualize JCHR computations (VisualCHR), and a generic search engine (JASE) to solve combinatorial problems (Chapter 6).
- Finally, two applications have been taken to study the benefit of rule-based constraint programming to solving real problems in timetabling and room assignment (Chapter 7).

Tacking all techniques together, the research presented in this paper enables us to make constraint solving easier, allows for a declarative problem representation, and can be implemented efficiently.

Bibliography

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer-Verlag, November 1997.

[2] S. Abdennadher. *Analyse von regelbasierten Constraintlösern (in German)*. PhD thesis, Computer Science Institute, LMU Munich, 1998.

[3] S. Abdennadher and H. Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Flexible Query Answering Systems*. Springer-Verlag, 2000.

[4] S. Abdennadher and T. Frühwirth. On completion of constraint handling rules. In *4th International Conference on Principles and Practice of Constraint Programming, CP98*, LNCS 1520. Springer-Verlag, 1998.

[5] S. Abdennadher and T. Frühwirth. Operational equivalence of CHR programs and constraints. In *5th International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713. Springer-Verlag, 1999.

[6] S. Abdennadher, T. Frühwirth, and H. Meuss. On confluence of constraint handling rules. In *2nd International Conference on Principles and Practice of Constraint Programming, CP96*, LNCS 1118. Springer-Verlag, August 1996.

[7] S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4(2), May 1999.

[8] S. Abdennadher and M. Marte. University timetabling using constraint handling rules. In *Actes des Journées Francophones de Programmation en Logique et Programmation par Contraintes*, 1998.

[9] S. Abdennadher and M. Marte. University course timetabling using constraint handling rules. *Journal of Applied Artificial Intelligence, Special Issue on Constraint Handling Rules*, 2000.

[10] S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In *6th International Conference on Principles and Practice of Constraint Programming, CP00*, LNCS 1894. Springer-Verlag, 2000.

[11] S. Abdennadher and C. Rigotti. Using confluence to generate rule-based constraint solvers. In *Third International Conference on Principles and Practice of Declarative Programming*. ACM Press, September 2001. To appear.

[12] S. Abdennadher, M. Saft, and S. Will. Classroom assignment using constraint logic programming. In *The Second International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming*, 2000.

[13] S. Abdennadher and H. Schütz. Model generation with existentially quantified variables and constraints. In *6th International Conference on Algebraic and Logic Programming*, LNCS 1298. Springer-Verlag, 1997.

[14] S. Abdennadher and H. Schütz. $\mathrm{CHR}^{\vee}$: A flexible query language. *Flexible Query Answering Systems*, LNAI 1495, 1998.

[15] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling*, 17(7):57–73, 1993.

[16] A. Aggoun, F. Bueno, M. Carro, P.Deransart, M. Fabris, W. Drabent, G. Ferrand, M. Hermenegildo, C. Lai, J. Lloyd, J. Maluszynski, G. Puebla, and A. Tessier. CP Debugging Needs and Tools. In *International Workshop on Automated Debugging*, pages 103–122, 1997.

[17] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216. ACM Press, 1993.

[18] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of ACM*, 26(11):832–843, 1983.

[19] The ANTLR translator generator. Internet: http://www.antlr.org, 1999.

[20] K. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *5th International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713. Springer-Verlag, 1999.

[21] K.R. Apt. Some remarks on boolean constraint propagation. In *New Trends in Constraints*. Lecture Notes in Artificial Intelligence 1865, 2000.

[22] F. Azevedo and P. Barahona. Timetabling in constraint logic programming. In *Proceedings of 2nd World Congress on Expert Systems*, 1994.

[23] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of the 15th International Conference on Data Engineering*, pages 188–197. IEEE Computer Society, 1999.

[24] C. Bessière and J. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Second International Conference on Principles and Practice of Constraint Programming*, LNCS 1118, pages 61–75. Springer, 1996.

[25] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.

[26] Alan Borning, Bjorn N. Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.

[27] F. Bry and A. Yahya. Minimal model generation with positive unit hyper-resolution tableaux. In *5th Workshop on Theorem Proving with Tableaux and Related Methods*, Springer LNAI, 1996.

[28] M. Carter and C. Tovey. When is the classroom assignment problem hard? *Operations Research*, 40(1):28–39, 1989.

[29] Y. Caseau, F. Josset, and F. Laburthe. Claire: Combining sets, search, and rules to better express algorithms. In *ICLP99*, 1999.

[30] T. Chew, M. Henz, and K. Ng. A toolkit for constraint-based inference engines. In *Practical Aspects of Declarative Languages*, 2000.

[31] H. Christiansen. Automated reasoning with a constraint-based metainterpreter. *Journal of Logic Programming*, 37(1-3):213–254, 1998. Special issue on Constraint Logic Programming.

[32] H. Christiansen and D. Martinenghi. Symbolic constraints for meta-logic programming. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 2000.

[33] Constraint Handling Rules Online,
`http://www.pms.informatik.uni-muenchen.de/~webchr/`

[34] A. Chun. Constraint programming in java with JSolver. In *Practical Application of Constraint Logic Programming*, 1999.

[35] K. Clark. *Logic and Databases*, chapter Negation as Failure, pages 293–322. Plenum Press, 1978.

[36] P. Codognet and D. Diaz. Boolean constraint solving using clp(FD). In D. Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, Vancouver, Canada, 1993. The MIT Press.

[37] J. Cohen. A view of the origins and development of Prolog. *Communications of the ACM*, 31(1):26–36, 1988.

[38] H. Decker. An extension of SLD by abduction and integrity maintenance for view updating in deductive databases. In *Proc. of JICSLP'96*, pages 157–169, 1996.

[39] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. Technical Report TR-LP-37, ECRC, Munich, Germany, May 1988.

[40] R. Eckstein, M. Loy, and D.Wood. *Java Swing.* O'Reilly, 1998.

[41] S. Etalle, M. Gabrielli, and M. Meo. Unfold/fold transformations of CCP programs. In *9th International Conference on Concurrency Theory*, 1998. Corrected version.

[42] J. Ferland and S. Roy. Timetabling problem for university as assignment of activity to resources. *Computers and Operational Research*, 12(2):207–218, 1985.

[43] H. Frangouli, V. Harmandas, and P. Stamatopoulos. UTSE: Construction of optimum timetables for university courses — A CLP based approach. In *Proceedings of the Third International Conference on the Practical Applications of Prolog*, pages 225–243, 1995.

[44] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21–70, 1992.

[45] T. Frühwirth. *A Declarative Language for Constraint Systems: Theory and Practice of Constraint Handling Rules.* Habilitation, Computer Science Institute, LMU Munich, 1998.

[46] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.

[47] T. Frühwirth. Proving termination of constraint solver programs. In *New Trends in Constraints.* LNAI 1865, 2000.

[48] T. Frühwirth and S. Abdennadher. *Constraint-Programmierung: Grundlagen und Anwendungen.* Springer-Verlag, September 1997.

[49] T. Frühwirth and P. Brisset. High-level implementations of constraint handling rules. Technical report, ECRC, 1995.

[50] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming: An informal introduction. In G. Comyn, N.E. Fuchs, and M.J. Ratcliffe, editors, *Logic Programming in Action*, LNCS 636. Springer-Verlag, 1992.

[51] M. Gabbrielli, G. Levi, and M. Chiara Meo. Observable behaviors and equivalences of logic programs. *Information and Computation*, 122(1):1–29, October 1995.

[52] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[53] M. Henz and J. Würtz. Using Oz for college time tabling. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling*, pages 283–296, 1995.

[54] C. Holzbaur and T. Frühwirth. A prolog constraint handling rules compiler and runtime system. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 2000.

[55] J. Jaffar and J. Lassez. Constraint logic programming. In Michael J. O'Donnell, editor, *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, January 1987.

[56] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20, 1994.

[57] A. C. Kakas and P. Mancarella. Database updates through abduction. In *Proc. 16th Int'l Conf. on Very Large Databases*, pages 650–661. Morgan Kaufmann, California, 1990.

[58] A. C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 399–416, Cambridge, June 13–18 1995. MIT Press.

[59] C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In *Proceedings of the First Workshop on Principles and Practice of Constraints Programming*. MIT Press, April 1993.

[60] H. Kirchner and C. Ringeissen. A constraint solver in finite algebras and its combination with unification algorithms. In *Proc. Joint International Conference and Symposium on Logic Programming*, pages 225–239. MIT Press, 1992.

[61] S.C. Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, New Jersey, 1950.

[62] R. Kowalski and G. Wetzel F. Toni. Executing suspended logic programs. *Special Issue of Fundamenta Informaticae*, 34(3), 1998.

[63] E. Krämer. A generic search engine for a java constraint kit. Master's thesis, Ludwig-Maximilians-University, 2001.

[64] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1), 1992.

[65] A. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. Wiley, 1992. Volume 1, second edition.

[66] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[67] M. J. Maher. Equivalences of logic programs. In *Proceedings of Third International Conference on Logic Programming*, Berlin, 1986. Springer.

[68] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*. The MIT Press, May 1987.

[69] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *9th Int. Conf. on Automated Deduction (CADE)*, LNCS 310, Argonne, IL, USA, may 1988. Springer-Verlag.

[70] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[71] M. Marte. Constraint-based grammar school timetabling – A case study. Diplomarbeit, Lehr- und Forschungseinheit für Programmier- und Modellierungssprachen, Institut für Informatik, Ludwig-Maximilians-Universität München, 1998.

[72] M. Meier. Debugging constraint programs. *Lecture Notes in Computer Science*, 976:204–221, 1995.

[73] H. Meyer auf'm Hofe. ConPlan/SIEDAplan: Personnel assignment as a problem of hierarchical constraint satisfaction. In *Proceedings of the 3rd International Conference on the Practical Application of Constraint Technology*, pages 257–272, 1997.

[74] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 651–656. Pitman Publishers, 1988.

[75] S. Muggleton and L. De Raedt. Inductive Logic Programming : theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[76] Group of Prof. Dr. Bernd Krieg-Brückner. The graph visualization system davinci. `www.informatik.uni-bremen.de/daVinci/`.

[77] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46, 1999.

[78] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 102–106, Vienna, Austria, August 1992. John Wiley & Sons.

[79] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[80] T. Le Provost and M. Wallace. Generalised constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16(3):319–359, 1993.

[81] D. A. Randell, Z. Cui, and A. G. Cohn. A spatial logic based on regions and connection. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 165–176, Cambridge, MA, October 1992. Morgan Kaufmann.

[82] C. Ringeissen. Etude et implantation d'un algorithme d'unification dans les algèbres finies. Rapport de DEA, Université de Nancy I, 1990.

[83] C. Ringeissen and E. Monfroy. Generating propagation rules for finite domains via unification in finite algebra. In *ERCIM Working Group on Constraints / CompulogNet Area on Constraint Programming Workshop*, 1999.

[84] K. Ryall, J. Marks, and S. Shieber. An interactive constraint-based system for drawing graphs. In *User Interface Software and Technology (*, 1997.

[85] D. Sabin and E. C. Freuder. Understanding and improving the mac algorithm. In *Third International Conference on Principles and Practice of Constraint Programming*, LNCS 1330, pages 167–181. Springer, 1997.

[86] V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, Cambridge, 1993.

[87] V. A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages.* ACM Press, January 1991.

[88] A. Schaerf. A survey of automated timetabling. Technical Report CS-R9567, CWI - Centrum voor Wiskunde en Informatica, 1995.

[89] M. Schmauss. A constraint library for java. Master's thesis, Ludwig-Maximilians-University, 1999.

[90] C. Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. MIT Press, Cambridge, MA, USA.

[91] H. Simonis and A. Aggoun. Search-tree visualisation. In *Debugging Systems for Constraint Programming.* LNCS 1870, Springer Verlag, 2000.

[92] D. A. Smith. Mixlog: A generalized rule based language. In *VIèmes Journées Francophones de Programmation en Logique et programmation par Contraintes.* Hermes, 1997.

[93] G. Smolka. The Oz programming model. *Lecture Notes in Computer Science*, 1000, 1995.

[94] G. Smolka and C. Schulte. Logische Programmierung. Skriptum zur Vorlesung, Fachbereich 14 - Informatik, Universit"at des Saarlandes, 1993.

[95] H. Toivonen, M. Klemettinen, P. Ronkainen, K. Hätönen, and H. Mannila. Pruning and grouping of discovered association rules. In *Workshop Notes of the ECML-95 Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases*, pages 47–52, April 1995.

[96] M. Torrens, R. Weigl, and B. Faltings. Java Constraint Library: bringing constraint technology to the internet using the java language. In *Working Notes of the Workshop on Constraints and Agents*, 1997.

[97] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Massachusetts, 1989.

[98] P. van Hentenryck. Constraint logic programming. *The Knowledge Engineering Review*, 6, 1991.

[99] P. van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1-3), December 1992.

[100] M. Wallace. Practical applications of constraint programming. *Constraints Journal*, 1(1,2), September 1996.

[101] G. Wetzel and F. Toni. Semantic query optimization through abduction and constraint handling. *Flexible Query Answering Systems*, LNAI 1495:366–381, 1998.

[102] J. E. Wunderwald. *Adding Bottom-up Evaluation to Prolog*. PhD thesis, Technische Universität München, 1996.

[103] N. Zhou, S. Kaneko, and K. Yamauchi. DJ: A java-based constraint language and system. In *Proceedings of the Annual JSSST Conference*, 1998.

Appendix A

Horn Clause Programs and SLD Resolution

Even though we expect the reader to be familiar with Horn clause programs and SLD resolution, we give some definitions in order to introduce our terminology and notation, which we will need for comparisons with $\mathrm{CHR}^\vee$.

A.1 Syntax

A *Horn clause program* is a set of *Horn clauses*, also called *rules*, which are formulas of the form $H \leftarrow B$, where H is an *atom*, i.e., an atomic first-order formula and B is a conjunction of atoms. We call H the *head* and B the *body* of the rule.

A.2 Declarative Semantics

The logical meaning of a Horn clause program P is given by its *completion* [35]: The *completed definition* of a predicate p is a formula of the form $\forall \bar{v}(p(\bar{v}) \leftrightarrow (\exists \bar{x}_1(\bar{v}{=}\bar{t}_1 \wedge B_1) \vee \ldots \vee \exists \bar{x}_n(\bar{v}{=}\bar{t}_n \wedge B_n)))$,[1] where $p(\bar{t}_1) \leftarrow B_1, \ldots, p(\bar{t}_n) \leftarrow B_n$ are all the clauses with head predicate p in P, every $\bar{x}_i$ is the list of variables occurring in the ith such clause, and $\bar{v}$ is a list of fresh variables of appropriate length. The *completion* of P consists of the completed definitions of all the predicates occurring in P and a theory defining $=$ as syntactic equality.

A.3 Operational Semantics

The operational semantics can be described as a state transition system for *states* of the form G, where G (the *goal*) is a conjunction of atoms. Transitions from a state G are possible if for some fresh variant[2] of a rule $H \leftarrow B$ in the given program P and some atom A in the goal G the head H and A are unifiable.

[1] Here $\bar{v}{=}\bar{t}_i$ stands for the conjunction of equations between respective components of the lists $\bar{v}$ and $\bar{t}_i$.

[2] Two formulas or terms are variants, if they can be obtained from each other by a variable renaming. A fresh variant contains only new variables.

Unfold	
If	$(H \leftarrow B)$ is a fresh variant of a rule in P
and	θ is a most general unifier of H and A
then	$A \wedge G \mapsto (B \wedge G)\theta$

Figure A.1: SLD resolution step

In the resulting state, A is replaced by B and a most general unifier of H and A is applied to $B \wedge G$.

This computation step is also given in Figure A.1. Conjunctions are considered to be associative and commutative.

We are looking for chains of such transitions from the initial state, which consists of the user-supplied query to some final state, where the goal must be the empty conjunction $\top$.

Given some state $A \wedge G$, there are two degrees of nondeterminism when we want to reduce it to another state:

- Any atom in the conjunction $A \wedge G$ can be chosen as the atom A.
- Any rule $(H \leftarrow B)$ in P for which H and A are unifiable can be chosen.

In order to achieve certain completeness properties of SLD resolution we have to try all possibilities w.r.t. the second degree. But it is a fundamental property of SLD resolution that w.r.t. the first degree it suffices to choose an arbitrary atom A. So the first degree of nondeterminism is of "don't-care" type, while the second is of "don't-know" type.

We leave the don't-care nondeterminism implicit in the calculus as usual, whereas we note the don't-know nondeterminism in a refined version of the **Unfold** computation step in Figure A.2. With this modified rule we construct

UnfoldSplit	
If	$(H_1 \leftarrow B_1), \ldots, (H_n \leftarrow B_n)$ are fresh variants of all those rules in P for which H_i $(1 \leq i \leq n)$ is unifiable with A θ_i is a most general unifier of H_i and A $(1 \leq i \leq n)$
then	$A \wedge G \mapsto (B_1 \wedge G)\theta_1 \mid \ldots \mid (B_n \wedge G)\theta_n$

Figure A.2: SLD resolution step with case splitting

trees of states rather than sequences of states. The root is the initial state as given above. Every inner node has children according to some application of **UnfoldSplit**. Leaves are either *successful* leaves, where the goal has become empty, or *failing* leaves if there are no appropriate rules in P for the chosen goal atom A.

List of Figures

Index