Rule-Based Constraint Propagation

Theory and Applications

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof.mr. P.F. van der Heijden
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op vrijdag 3 december 2004, te 14.00 uur

door

Sebastian Brand

geboren te Erfurt, Duitsland.

Promotiecommissie:

Promotor

prof.dr. K. R. Apt

Overige leden

prof.dr. J. F. A. K. van Benthem
prof.dr. M. van Lambalgen
prof.dr. E. Monfroy
prof.dr. M. de Rijke
dr. S. Etalle
dr. Zs. M. Ruttkay
dr. L. Torenvliet

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Printed and bound by Ipskamp, Enschede.

ISBN: 90–6196–526–8

Contents

Chapter 1

Introduction

The notion of the constraint satisfaction problem (CSP) provides a general framework for formulating problems. In this framework, a problem solution corresponds to a variable assignment. The problem context is formalised by stating the range of values the variables can assume, and by specifying for some subsets of the variables which combinations of their values are acceptable — in other words, which constraints must be met.

A large variety of problems can be modelled as CSPs, and in many cases the models are very natural. Consider the graph colouring problem: we wish to colour the vertices of a graph in such a way that connected vertices differ in their colour. In a straightforward formulation as a CSP, each vertex corresponds to a variable ranging over the colours, and two connected vertices give rise to a disequality constraint on the respective variables. The term unification problem can be viewed as a CSP in which the variables range over a term universe and are constrained by equalities. A propositional formula in conjunctive normal form can be seen as a CSP by regarding each clause as a constraint on its Boolean variables. More generally, the CSP framework is applicable to problems in many areas, including Artificial Intelligence (temporal and spatial reasoning, computer vision, planning, computational logic), Operations Research (scheduling, time-tabling, resource allocation), and Bio-informatics (protein structure reconstruction, sequence alignment).

The expressiveness of the CSP framework has consequences for the solving algorithms. Solving CSPs with finite variable domains is NP-complete in general, and one should therefore not expect computationally tractable algorithms. A general method to find solutions of a CSP consists of search, that is, the CSP is split into subproblems which are considered separately. For instance, every assignment of a variable to a domain value induces a subproblem.

In the constraint programming approach to solving CSPs, search is combined with *constraint propagation* to reduce the search space. The principle of constraint propagation is controlled inference: from the available constraints and domains, certain new constraints or smaller domains are inferred. In this way, a CSP is transformed by making selected implicit information explicit. Usually, the result of propagation is characterised as a form of *local consistency*. Constraint propagation is often a very cost-effective method to reduce the problem solving time, insofar as more time is saved in search than spent on propagation.

The question that arises is how constraint propagation can be described and implemented. Conventionally, this is done by generic or specialised constraint propagation algorithms, implemented in an imperative programming style. Generic algorithms, by definition, do not take into account the structure of specific constraints. Constraint-specific algorithms, on the other hand, typically require considerable expertise in the development of constraint propagation algorithm and are hard to understand or verify.

In this thesis, we advocate a *rule-based view* on constraint propagation.

Rule-Based Constraint Propagation

Rule-based programming means the formulation of programs in terms of rules, i. e. premise–conclusion pairs. Such programs are executed by a repeated application of the rules. Hence, rule-based programming is declarative: the program logic is separated from the control of the execution. The interest in rule-based computation goes back at least to the 1970s, when production rule systems were extensively studied in Artificial Intelligence.

We apply the rule-based paradigm to constraint propagation and consequently consider *constraint propagation rules*. Our notion of a constraint propagation rule is very basic:

$$\mathcal{A} \rightarrow \mathcal{B}$$

is a constraint propagation rule if $\mathcal{A}, \mathcal{B}$ are sets of constraints. Here are some examples of such rules:

$$x > y,\ y > z \quad\rightarrow\quad x > z \tag{r_1}$$

$$\mathsf{and}(x, y, z),\ \mathsf{or}(y, z, w),\ w = 1 \quad\rightarrow\quad y = 1,\ x = z \tag{r_2}$$

$$\mathsf{rcc8}(x, y, z), x \in \{\mathsf{disjoint}, \mathsf{inside}\}, z \in \{\mathsf{contains}, \mathsf{equal}\} \ \rightarrow\ y \neq \mathsf{covers} \tag{r_3}$$

The rule r_1 captures transitivity of the ordering relation $>$ viewed as a constraint. r_2 propagates a fact about the constraints and and or which model the respective logical operators. Spatial knowledge is expressed in rule r_3: x, y, z are the topological relations of the region pairs (A, B), (B, C), (A, C), respectively.

The formulation of constraint propagation in terms of rules offers several advantages. Since propagation rules are declarative, the *correctness* of the constraint propagation step represented by a rule can be verified directly and per rule with the definitions of the involved constraints. Rules represent *directed* knowledge, and in that way they control inference. The local consistency established by a set of rules can be examined independently of the concrete rule scheduling algorithm.

Contributions and Overview

We argue in this dissertation that a rule-based approach to constraint propagation is useful for both explaining and implementing it. We do so by paying attention to theoretical aspects of rule-based constraint propagation as well as to applications in constraint programs. When discussing the applications, in line with the constraint programming approach to problem solving, we focus especially on modelling declaratively. In detail, we discuss the following topics.

The three chapters following the introductory Chapter 2 on rule-based constraint programming are devoted to problem-independent issues involving constraint propagation rules. In all cases, we discuss in detail a class of constraint propagation rules of particular interest, the *membership rules*.

Schedulers for constraint propagation rules. In Chapter 3, we consider the problem of computing with constraint propagation rules. We start from the completely general view of constraint propagation as fixpoint computation of functions, and review a corresponding generic iteration algorithm.

We revise this algorithm with a *dynamic modification* of the set of iterated functions. Specifically, we provide conditions for the removal of functions from this set, so as to improve convergence of the fixpoint computation. The benefit of this technique is multiplied if one deals with sequences of fixpoint computations, as is the case in constraint programming in which constraint propagation is executed repeatedly. A dynamic reduction in the function set then helps convergence in all later computations.

By implementing the revised iteration algorithm for concrete sets of membership rules, we demonstrate the viability of this way of performing constraint propagation. Furthermore, by an empirical evaluation we find that the revised rule scheduler performs very well in comparison with the generic scheduler as well as with the scheduler used in an implementation of `CHR`, a language specifically designed for rule-based constraint propagation.

This chapter is based on a collaboration with Krzysztof Apt, which appeared as [Apt and Brand, 2003]. A combination with the following chapter will appear as [Brand and Apt, 2005].

Redundancy in constraint propagation rule sets. In Chapter 4, we turn to the question whether *each* propagation rule in a set of rules is needed for the result of propagation. A natural characterisation of the local consistency established by a set of rules is based on their common fixpoints (where rules are viewed as functions in an abstract setting). Consequently, we formulate the notion of *redundancy of a rule with respect to a rule set* as follows: removing a redundant rule from the set does not change the common fixpoints. This leads to the notion of a *minimal* rule set, which contains no redundant rules.

We also investigate rule redundancy empirically, with the help of an implementation of minimisation. In recent years, a number of methods for the automatic generation of classes of constraint propagation rules have been published. While all of these rule generation methods strive to generate rule sets that are minimal in a sense, they fall short on rigour or generality. By processing a number of concrete rule sets generated by such methods, we find that many of the sets are not minimal. We provide here a redundancy notion that is theoretically well-founded, comprehensive, and feasible.

This chapter is an extended version of [Brand, 2003].

Incremental generation of constraint propagation rules. In Chapter 5, we approach the problem of generating propagation rules *incrementally*. By this, we mean automatic rule generation as transformations of rule sets into rule sets. One example is the combination of two rules to a new rule: $c_1 \to \mathcal{B}$ and $c_2 \to \mathcal{B}$ leads to $c_1 \vee c_2 \to \mathcal{B}$. The crucial requirement is that the disjunctive constraint $c_1 \vee c_2$ must be representable in the underlying language of the considered constraint propagation rules.

We then study incremental rule generation for the specific language of the membership rules. We regard rule sets associated with constraints, and consider the following question: suppose it is known how some given constraints relate to each other, then how do their associated rule sets relate to each other? The relations of constraints we are interested in are incremental constraint definitions, for example, separate constraints versus their conjunction.

For various such incremental constraint definitions, we explain how the associated membership rule sets are incrementally obtained. A natural question concerns the propagation associated with the respective rule sets. We give conditions on the input rule sets that allow us a characterisation of the propagation of the result rule set.

The usability of incremental rule generation for membership rules is demonstrated by an implementation and examples.

The material in this chapter is based on joint work with Eric Monfroy. It appeared as [Brand and Monfroy, 2003],

We then consider practical applications that we solve by constraint programming and rule-based constraint propagation.

Test pattern generation for sequential circuits. In Chapter 6, we consider a problem from electrical engineering. Since the production process of modern digital circuits is not error-free, it is necessary to verify produced circuits against their specifications. This comparison must be behavioural as the internal circuit structure is inaccessible. Therefore, *test patterns*, sequences of input data, are used to verify the circuit function by comparing observed and expected output. The test pattern generation problem concerns the generation of such tests for specific circuit faults.

We consider the case of *sequential* circuits, which have an internal state and for which test generation is thus substantially more complex than for combinational (stateless) circuits. While propositional logic seems natural in this domain, our modelling approach is based on *multi-valued* logics. The extra values are used for approximating the original problem and for carrying heuristic information.

We develop three different multi-valued logics and compare them by means of standard benchmarks, using our constraint-based implementation. The constraint propagation in our implementation is based on membership rules, and we apply the techniques introduced in the preceding chapters.

This chapter contains a completely rewritten version of [Brand, 2001b].

Modal satisfiability checking. Chapter 7 presents a constraint-based approach for deciding the satisfiability of modal logic formulas. One approach to solving this problem consists of reformulating it into sequences of propositional satisfiability problems. We extend it by employing a *three-valued* logic in the subproblems instead; the extra value reflects structural information that is lost in the propositional translation. The resulting subproblems are thus non-Boolean, and we view them as CSPs.

We describe a corresponding implementation, which relies on several forms of rule-based constraint propagation. We evaluate the approach and implementation using standard benchmarks. The results show that the three-valued constraint-based approach is competitive with, and in some instances superior to, the purely propositional approach. This shows the interest of a refined modelling made possible by the expressiveness of the CSP-framework.

This chapter reflects joint work with Rosella Gennari and Maarten de Rijke. It appeared as [Brand et al., 2004], but has been adapted for this dissertation.

We then return to an application-independent topic.

Array constraints. In Chapter 8, we consider constraints that naturally arise when information is arranged in arrays (matrices). We study two forms of constraint propagation for such *array constraints*. Starting from generic template rules that capture the desired local consistency, we derive specialised constraint propagation rules. We then design algorithms that embody the rules and their correctness conditions. The results are systematically developed constraint propagation algorithms for array constraints.

This chapter contains a fully revised and substantially extended version of [Brand, 2001a].

In the next two chapters, we study a domain in which information is naturally structured in array form.

Qualitative spatial reasoning with relation variables. In Chapter 9, we discuss an alternative constraint-based approach to qualitative spatial reasoning. In contrast to the standard approach, in which qualitative relations are viewed as constraints, we model relations as variables. These *relation variables* are arranged in an array.

This approach is particularly suitable for qualitative spatial reasoning since space has many aspects (topology, size, direction, etc.). The advantage of this view is that the properties of one spatial aspect as well as the integration of different spatial aspects are expressed as plain constraints on the relation variables. This makes specialised consistency algorithms redundant by a reduction to generic constraint propagation techniques, which we realise by rules.

This chapter appeared as [Brand, 2004].

Qualitative simulation. Chapter 10 reports our approach to qualitative reasoning involving *change*, based on constraints over relation variables. We use *temporal logic* to describe dynamic system behaviour, which allows concise statements of complex circumstances.

The temporal logic formulas are translated into constraints over the relation variables. We give one translation that simply unfolds the temporal and logical operators, and a second translation that retains the underlying structure by using the array constraints introduced in Chapter 8. This *array translation* leads to particularly compact CSPs and is much more amenable to constraint propagation.

We describe an implementation and discuss case studies.

This chapter reports joint and on-going work with Krzysztof Apt.

In Chapter 11, we summarise and contemplate future research issues.

Chapter 2

Rule-Based Constraint Programming

Rule-based constraint programming means the adoption of a rule-based approach to solving constraint satisfaction problems. In this chapter, we first introduce constraint programming and then discuss rule-based programming. We proceed by discussing the application of this paradigm to constraint programming, before coming to what we regard as its most relevant aspect, namely rule-based constraint propagation.

2.1 Constraint Programming

2.1.1 Overview

Constraint programming is an alternative approach to programming in which a problem is first modelled declaratively and then solved by general or domain-specific methods. See [Tsang, 1993, Marriot and Stuckey, 1998, Apt, 2003, Dechter, 2003, Frühwirth and Abdennadher, 2003], for instance.

We begin with an overview; the formal framework is introduced in the following sections.

Modelling. A problem model in constraint programming consists of requirements — constraints — on variables so that acceptable variable assignments correspond to solutions to the problem. The variables have domains, i. e. sets of possible values. A constraint is a relation that specifies which combination of domain values is acceptable; it can be defined extensionally or intensionally. A problem formulated in this way is called a *constraint satisfaction problem* (CSP).

The CSP framework is expressive; many computationally intractable problems can immediately be formulated as CSPs. In a 3-SAT problem, each clause constrains three propositional variables. In graph colouring, where different colours must be assigned to connected vertices, the constraints are simple disequalities. Combinatorial problems often have simple formulations as CSP. A strength of modelling with constraints is its flexibility due to the compositional nature of CSPs: individual constraints can be changed, added or removed disregarding the rest of the problem.

Furthermore, problems exist that do not directly correspond to CSPs but contain CSPs as subproblems. For instance, optimisation problems can often be naturally decomposed into constraints describing what a solution is, and an objective function rating the quality of a solution. This leads to the concept of a *constraint optimisation problem*. In other situations, a problem gives rise to a *sequence* of CSPs. An example of this is planning. The problem whether a plan of a given length exists transforming one state into another can typically be represented as a CSP. Shortest plans can be found by searching for fixed-length plans and iteratively increasing the plan length, which means repeatedly trying to solve a CSP.

Solving. For some specific CSP classes, specialised solution methods exist. A system of linear equations induces a CSP in which every equation is a constraint, and which can be solved well by Gaussian Elimination, for example. Term unification can be viewed as a CSP in which the variables range over some term universe; a unification algorithm solves such a problem.

A general method to find solutions of CSPs is systematic *search*: splitting the problem into more specific subproblems that are examined separately. Backtracking is a commonly used algorithm for this purpose.

The search space can be reduced by *constraint propagation*. The principle of constraint propagation is the *controlled deduction* of new constraints. The derived constraints are added to the problem so as to make its solutions *more explicit* without changing them. The options for constraint propagation include, on the one hand, which or how many currently explicit constraints to consider at a time, and on the other hand, what constraints to deduce. The identification of the types of constraint propagation that are both useful and have acceptable computational cost and the development of corresponding efficient *constraint propagation algorithms* are important issues in constraint programming research.

The result of complete constraint propagation is typically characterised by a *local consistency* notion. Generally, a local consistency only approximates global consistency (concerning the entire CSP), and usually the two are incomparable in the sense that none entails the other.

Since a CSP model of a problem does not prescribe the solution method, alternatives to systematic search are possible. Notably, *local search* turns out to perform very well on some types of CSPs. In local search, a total variable assignment — a solution *candidate* — is considered. If not all constraints are satisfied, the assignment is modified, and the process is repeated. The changes to the assignment are mostly local and controlled by heuristics. For instance, a variable is selected whose assigned value violates a constraint, and the assigned value is replaced.

In contrast to systematic search, local search is incomplete: it is not ensured that all possible assignments (i. e. the complete search space) are visited. Hence, local search is unusable for proving unsatisfiability, nor is it guaranteed to find a solution of a satisfiable problem.

A brief history. Constraint programming has its roots in the field of constraint satisfaction in Artificial Intelligence in the 1970s; see for example [Montanari, 1974, Kumar, 1992]. In the 1980s, these techniques were connected with the declarative problem solving approach of logic programming, which resulted in constraint logic programming [Jaffar and Maher, 1994]. More recently and on-going, a fruitful import of techniques notably from Operations Research has broadened the scope of the field. Applications of constraint programming include various problems in Artificial Intelligence (temporal reasoning, various forms of spatial reasoning) and Combinatorial Optimisation (scheduling, resource allocation, configuration, planning). Industrial interest in constraint programming techniques continues to provide a significant impetus.

2.1.2 Constraint Satisfaction Problems

We now introduce constraint programming formally.

Constraints

Consider a finite sequence of different variables

$$X = x_1, \ldots, x_m$$

with respective domains

$$D_1, \ldots, D_m,$$

so each x_i takes its value from the set D_i. A ***constraint*** C on X is a pair

$$\langle C_R, X\rangle.$$

C_R is an m-ary relation and a subset of the Cartesian product of the domains,

$$C_R \subseteq D_1 \times \cdots \times D_m.$$

The elements of C_R are the *solutions* of the constraint, and m is its *arity*. Nothing more is stipulated about C_R; in particular, it can be defined intensionally and it can be infinite.

It is useful to mention two special cases of constraints. In the true constraint, we find $C_R = D_1 \times \cdots \times D_m$, while $C_R = \varnothing$ in the false constraint. These are the only two cases in which we admit $m = 0$, whereas we generally require $m \geqslant 1$.

We sometimes write $C(X)$ for the constraint and often identify C with C_R.

2.1.1. Example. Here are some constraints and their variables.

$$\langle \{\langle \mathsf{A},\mathsf{B}\rangle, \langle \mathsf{C},\mathsf{A}\rangle, \langle \mathsf{B},\mathsf{C}\rangle\},\ \langle x,y\rangle \rangle, \qquad x,y \in \{\mathsf{A},\mathsf{B},\mathsf{C}\};$$

$$\langle \{ \langle a,b,c,n\rangle \in \mathbb{N}^4 \mid a^n + b^n = c^n \}, \langle x,y,z,u\rangle \rangle, \qquad x,y,z,u \in \mathbb{N};$$

$$\langle \{ \langle p_1,\ldots,p_m\rangle \in \{0,1\}^m \mid \exists k.\, p_k = 1 \},\ \langle x_1,\ldots,x_m\rangle \rangle, \quad x_i \in \{0,1\} \text{ for all } i.$$

$\mathbb{N}$ is the set of natural numbers. □

Constraint Satisfaction Problems

A ***constraint satisfaction problem***, in short CSP, consists of a finite sequence of variables $X = x_1,\ldots,x_n$ with respective domains $\mathcal{D} = D_1,\ldots,D_n$, and a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. A CSP can thus be viewed as a triple

$$\langle \mathcal{C}, X, \mathcal{D}\rangle.$$

We use also the notational variant $\langle \mathcal{C};\ x_1 \in D_1,\ldots,x_n \in D_n\rangle$.

Solutions

Consider some variable sequence $X = x_1,\ldots,x_n$ and an element $d = d_1,\ldots,d_n$ of the product of variable domains $D_1 \times \cdots \times D_n$. By the ***projection*** of d on a subsequence $Y = x_{i(1)},\ldots,x_{i(\ell)}$ of X, we mean the sequence $d_{i(1)},\ldots,d_{i(\ell)}$ which we denote by $d[Y]$. In particular, we have $d[x_k] = d_k$. Lifting this notion to constraints, we write $C[Y]$ for the set $\{ d[Y] \mid d \in C_R \}$ where $C = \langle C_R, X\rangle$.

By a ***solution*** to the CSP $\langle \mathcal{C}, X, \mathcal{D}\rangle$ we mean an element $d \in D_1 \times \cdots \times D_n$ such that for each constraint $C \in \mathcal{C}$ on a sequence of variables Y we have $d[Y] \in C$. We call a CSP ***consistent*** if it has a solution. The set of all solutions of a CSP $\mathcal{P}$ is denoted by $Sol(\mathcal{P})$.

2.1.2. Example. Consider the CSP

$$\mathcal{P} = \langle \{\mathsf{fermat}(x,y,z,n), \mathsf{even}(y)\};\ x,y,z,n \in \mathbb{N}\rangle.$$

The constraint $\mathsf{fermat}(x,y,z,n)$ is $x^n + y^n = z^n$, and even is the set of even integers. $\mathcal{P}$ is consistent: the tuple $d = \langle 3,4,5,2\rangle$ is a solution. Indeed, we have

$$d[x,y,z,n] = \langle 3,4,5,2\rangle \in \mathsf{fermat},$$
$$d[y] = 4 \in \mathsf{even}.$$

A variation of $\mathcal{P}$ is $\mathcal{P}' = \langle \{\mathsf{fermat}(x,y,z,2), \mathsf{even}(y)\};\ x,y,z \in \{1,\ldots,10\}\rangle$. We find

$$Sol(\mathcal{P}') = \{\langle 3,4,5\rangle, \langle 6,8,10\rangle, \langle 8,6,10\rangle\}$$

as the solution set of $\mathcal{P}'$. □

From now on, we use the interval expression $[a..b]$ with integers a,b to denote the integer set $\{ e \mid a \leqslant e \leqslant b \}$.

2.1.3 Solving CSPs by Search and Propagation

Equivalence

We view the search for solutions to a CSP as a process of transforming CSPs. This makes it necessary to relate CSPs to each other disregarding their representation. Instead, we use their solution sets, and accordingly define a notion of *equivalence*.

A natural definition for two CSPs on the same variables is to say that they are equivalent if they have exactly the same solutions. We extend this notion to sets of CSPs.

2.1.3. Definition. Assume a CSP $\mathcal{P}$ and CSPs $\mathcal{Q}_i$, $i \in [1..m]$, that are all on the same sequence of variable. If

$$Sol(\mathcal{P}) = \bigcup_{i \in [1..m]} Sol(\mathcal{Q}_i)$$

then we say that $\mathcal{P}$ is ***equivalent*** to the union of $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$. □

2.1.4. Example. The CSP $\mathcal{P} = \langle \mathcal{C};\ X;\ \mathcal{D} \rangle$ is equivalent to the union of

$$\begin{aligned}
\mathcal{Q}_1 &= \langle \mathcal{C} \cup \{\mathsf{even}(x)\};\ X;\ \mathcal{D} \rangle, \\
\mathcal{Q}_2 &= \langle \mathcal{C} \cup \{\mathsf{odd}(x), y \leqslant 10\};\ X;\ \mathcal{D} \rangle, \\
\mathcal{Q}_3 &= \langle \mathcal{C} \cup \{\mathsf{odd}(x), y > 10\};\ X;\ \mathcal{D} \rangle,
\end{aligned}$$

where $x, y \in X$ and odd has the expected meaning. □

Solving Algorithm Schema

We are now in a position to give a basic algorithm schema for finding solutions of CSPs using ***depth-first search***; see Fig. 2.1. Three procedures are used. Solve is the main control procedure, Split generates two sub-CSPs from a given CSP, and Propagate performs constraint propagation. Both Split and Propagate maintain equivalence. This means that Solve is correct and complete in the sense that it returns successfully with a solution if and only if one exists — if the computation terminates.

Here are some instances of the Split procedure. We consider the simple case of splitting the CSP $\mathcal{P}$ into just two subproblems $\mathcal{P}_1$ and $\mathcal{P}_2$.

- ***Domain splitting*** is a commonly used method. $\mathcal{P}, \mathcal{P}_1, \mathcal{P}_2$ differ only in the domain of some variable x. In $\mathcal{P}$, it is D_x.
  Domain partitioning: $D_x = D_{x,1} \,\dot{\cup}\, D_{x,2}$ is a partitioning. In $\mathcal{P}_1$ the domain of x is $D_{x,1}$ while in $\mathcal{P}_2$ it is $D_{x,2}$.
  Enumeration is a special case: we have $D_{x,1} = \{e\}$ and $D_{x,2} = D_x - \{e\}$, for some value $e \in D_x$.

$\boxed{\textsf{Solve}}$: CSP $\mathcal{P} \longmapsto \langle$solution sol, Boolean $success\rangle$

$\mathcal{P} := \textsf{Propagate}(\mathcal{P})$

if solution sol detected in $\mathcal{P}$ **then**
  return $\langle sol, \textsf{true}\rangle$
else if inconsistency detected in $\mathcal{P}$ **then**
  return $\langle \varnothing, \textsf{false}\rangle$
else
  $\mathcal{PS} := \textsf{Split}(\mathcal{P})$
  repeat
    choose and remove $\mathcal{P}$ from $\mathcal{PS}$
    $\langle sol, success\rangle := \textsf{Solve}(\mathcal{P})$
  until $success$ **or** $\mathcal{PS} = \varnothing$
  return $\langle sol, success\rangle$
end

$\boxed{\textsf{Split}}$: CSP $\mathcal{P} \longmapsto \langle$CSPs $\mathcal{P}_1, \ldots, \mathcal{P}_n\rangle$

// *$\mathcal{P}$ is equivalent to the union of $\mathcal{P}_1, \ldots, \mathcal{P}_n$, $n \geqslant 2$*

$\boxed{\textsf{Propagate}}$: CSP $\mathcal{P} \longmapsto$ CSP $\mathcal{P}'$

// *$\mathcal{P}'$ is equivalent to $\mathcal{P}$ but possibly "simpler"*

Figure 2.1: CSP solver schema with depth-first search and propagation

- ***Constraint splitting:*** Some constraints have an obvious disjunctive form, which affords splitting. Suppose $|x| = y$ occurs in $\mathcal{P}$. We can obtain the CSPs $\mathcal{P}_1, \mathcal{P}_2$ by replacing $|x| = y$ in $\mathcal{P}$ with $x = y$ and $-x = y$, resp.

The choices that are made when splitting generally have great influence on the solving performance. Consequently, heuristics have been developed to guide the splitting process. In the case of enumeration, one needs to decide which variable to enumerate, and in which order the domain values are tried. The general ***first-fail*** heuristic often performs well: it chooses the variable that is most constrained, e. g., has a smallest domain [Haralick and Elliott, 1980]. Specialised heuristics are used in many application domains.

2.1.4 Constraint Propagation and Local Consistency

The Propagate procedure of Fig. 2.1 is the most interesting one for us. Constraint propagation aims at transforming the CSP into an equivalent one that can be solved easier. Constraint propagation can thus be viewed as deduction, and the task is to control it in such a way that the computational cost is low but the inferred knowledge – in the form of constraints – is useful for subsequent propagation or search.

Constraint propagation is generally characterised by the properties of the resulting CSP. ***Local consistency*** notions are used for this purpose. The term 'local' reflects the observation that constraint propagation usually does not establish global consistency.

The most important local consistency notions in this work are generalised arc-consistency, bounds-consistency, and path-consistency. We define them now.

Generalised Arc-Consistency

Constraint propagation takes place for one constraint at a time in this case. The aim is to obtain the 'smallest possible' variable domains; in other words, to derive all *variable-value* disequality constraints. (Throughout this work, we use the term "disequality" for $\neq$ and "inequality" for $>, \geqslant, \leqslant$ and $<$.)

2.1.5. Definition. The constraint $C(x_1, \ldots, x_n)$ is ***generalised arc-consistent (GAC)*** if

for all x_i, $i \in [1..n]$, and all $e \in D_i$ we have $e \in C[x_i]$.

Recall that $C[x_i]$ stands for $\{\, d[x_i] \mid d \in C \,\}$. A CSP is generalised arc-consistent if each of its constraints is [Mohr and Masini, 1988]. □

In short, every domain value must participate in a local solution. The atomic step leading to GAC is: if there is some $e \in D_i$ for some x_i such that $e \notin C[x_i]$, then the domain is reduced by $D_i := D_i - \{e\}$.

2.1.6. Example. Consider the CSP

$$\mathcal{P} = \langle \{x^2 + y^2 = z^2, \mathsf{even}(y)\};\ x, y, z \in [1..10] \rangle.$$

$\mathcal{P}$ is not generalised arc-consistent, since for the variable y and $9 \in D_y$ we find $9 \notin \mathsf{even}[y]$. GAC-enforcing propagation of the constraint even must thus infer the constraint $y \neq 9$, or equivalently reduce the domain of y by $D_y := D_y - \{9\}$. In fact, propagating the constraint $\mathsf{even}(y)$ leads to $y \in \{2, 4, 6, 8, 10\}$.

Complete GAC-enforcing constraint propagation in $\mathcal{P}$ results in

$$\mathcal{P}' = \langle \{x^2 + y^2 = z^2, \mathsf{even}(y)\};\ x \in \{3, 6, 8\}, y \in \{4, 6, 8\}, z \in \{5, 10\} \rangle,$$

which is equivalent to $\mathcal{P}$ but makes the solutions more explicit. □

GAC on Conjunctive Constraints. Here is a fact about generalised arc-consistency that is useful in several of the following chapters.

2.1.7. LEMMA. *Consider two constraints C_1, C_2 that share at most one variable. If C_1 and C_2 are generalised arc-consistent individually, then the conjunctive constraint $C = C_1 \wedge C_2$ is generalised arc-consistent.*

PROOF. Suppose $C_1(X_1, y)$ and $C_2(X_2, y)$, sharing only the variable y, are generalised arc-consistent. Clearly, for each value in D_y there is a solution d_1 of C_1 and d_2 of C_2, respectively. We can form a solution d of C by just requiring $d[X_1] = d_1$ and $d[X_2] = d_2$, since X_1 and X_2 are disjoint. So every value in D_y can be extended to a solution of C. The remaining cases are straightforward. □

Generalised arc-consistency is a strong local consistency notion. It may also be computationally expensive: in general, the cost of establishing it on a constraint is exponential in the arity of the constraint. Sometimes computationally cheaper constraint propagation toward a weaker local consistency is more useful.

Bounds-Consistency

The following local consistency notion just checks the bounds of a domain, instead of every contained value.

2.1.8. DEFINITION. Assume that $C(x_1, \ldots, x_n)$ is a constraint such that each of its variables x_i has a totally ordered domain in which $\min(D_i)$ and $\max(D_i)$ are defined accordingly. C is ***bounds-consistent (BC)*** if

for all x_i we have $\min(D_i) \in C[x_i]$ and $\max(D_i) \in C[x_i]$.

□

2.1.9. EXAMPLE. Consider again

$$\mathcal{P} = \langle \{x^2 + y^2 = z^2, \mathsf{even}(y)\};\ x, y, z \in [1..10] \rangle.$$

The equivalent CSP

$$\mathcal{P}' = \langle \{x^2 + y^2 = z^2, \mathsf{even}(y)\};\ x \in \{3..8\}, y \in \{4..8\}, z \in \{5..10\} \rangle,$$

is bounds-consistent. □

Bounds-consistency is entailed by generalised arc-consistency, but is usually cheaper to establish. A significant representational benefit of using BC instead of GAC is that interval domains *remain* intervals: establishing BC cannot result in 'holes' in the domains. Intervals require little space to be represented, in contrast to unrestricted sets.

Path-Consistency

Finally, we introduce a local consistency notion considering multiple constraints at a time.

2.1.10. Definition. A CSP of only binary constraints is ***path-consistent (PC)*** [Montanari, 1974] if for every triple of variables x, y, z we have

$$C(x, z) = \{ (a, c) \mid b \text{ exists s.t. } (a, b) \in C(x, y) \text{ and } (b, c) \in C(y, z) \}.$$

It is assumed here that a unique constraint $C(u, w)$ for each pair of variables u, w exists, and that $C(u, w) = C^{-1}(w, u)$. By C^{-1} we mean the inverse relation of the binary relation C. □

In contrast to the cases of generalised arc-consistency and bounds-consistency, establishing path-consistency may require modification of the *constraints*, while the *variable domains* remain the same.

2.1.11. Example. Consider a simple graph-colouring problem. The corner points of a rectangle are to be coloured in red or green; connected corners must have different colours. In our CSP formulation we have a variable for each vertex, ranging over the colours, and disequality constraints for connected vertices:

$$\begin{aligned}
&\mathcal{P} = \langle \mathcal{C};\ v_1, \ldots, v_4 \in D \rangle,\\
&D = \{\mathsf{red}, \mathsf{green}\},\\
&\mathcal{C} = \{v_1 \neq v_2,\ v_2 \neq v_3,\ v_3 \neq v_4,\ v_4 \neq v_1\}.
\end{aligned}$$

We kept implicit the `true` constraints between the two unconnected vertices, namely $\langle D^2, \langle v_1, v_3 \rangle \rangle$ and $\langle D^2, \langle v_2, v_4 \rangle \rangle$. $\mathcal{P}$ is not path-consistent. Replacing $\mathcal{C}$ in $\mathcal{P}$ by

$$\mathcal{C}' = \mathcal{C} \cup \{v_1 = v_3,\ v_2 = v_4\},$$

we obtain a CSP $\mathcal{P}'$ that is equivalent to $\mathcal{P}$ and path-consistent. □

Path-consistency plays a central role in the view on qualitative temporal and spatial reasoning in which binary qualitative relations are represented as constraints.

2.2 Rule-Based Programming

When a program is a set of rules and the computation process consists of a repeated application of the rules, we speak of rule-based programming. By a rule we understand a premise–conclusion pair.

Rules can be found in various places in computer science. Automata in theoretical computer science are based on transition functions, which we can view as sets of rules in the form $s_1, a \rightarrow s_2$ where s_1, s_2 are states and a is an input symbol. Reasoning systems in computational logic are quite naturally rule-based, being based on logical calculi that consist of rules over logic formulas. In the field of term rewriting, rules are used to implement directed equational reasoning.

As a concrete programming paradigm, the rule-based approach received much attention in the 1970s with the rise of ***production systems*** in Artificial Intelligence. A production rule operates on the elements of the *working memory* of a production system and describes how they are changed. This development led to the general-purpose language OPS5 [Forgy, 1981] used for programming expert systems.

Logic Programming is a second rule-based formalism from that time (see e. g. [Lloyd, 1987]). It is realised in the Prolog language. A rule (clause) in a logic program relates an atomic formula in the conclusion (the clause head) with a sequence of atomic formulas in the premise (the clause body). The rules in a program are used to prove a goal, and compute a result in the form of a substitution in the process.

Interestingly, term ***unification***, which is at the core of logic programming systems, is itself amenable to a rule-based view. The Martelli-Montanari unification algorithm comprises six rules that can be used to decide whether a set of term equations has a unifying substitution. In the affirmative case, the algorithm yields a most general such substitution [Martelli and Montanari, 1982].

It is instructive to note the classification of rule-based systems with respect to ***forward chaining*** and ***backward chaining*** approaches. In a forward chaining system, of which production systems are an instance, inference adds derived information to simplify the problem. In a backward chaining system, such as Prolog, the reasoning starts from the goal and, via the rules, attempts to find facts supporting a proof. The rules embody a case distinction, and backtracking is used to explore the cases.

2.2.1 Rule-Based Constraint Programming

Constraint logic programming (CLP) originated from logic programming in the 1980s [Jaffar and Maher, 1994]. Hence, also CLP languages as such are rule-based. The observation that the unification operation in logic programming is just a special case of constraint solving led to the CLP(X) scheme [Jaffar and Lassez, 1987], in which X represents the domain of constraint solving (term equalities, arithmetic constraints over reals, finite domain constraints, ect.). In this light, the Martelli-Montanari unification algorithm which solves term equations is the first rule-based constraint solving method.

Another path to rule-based constraint programming originated in the area of term rewriting. The language ELAN [Borovanský et al., 1998] implements an approach to computation and deduction based on conditional rewrite rules and controlled by strategies. Its application to constraint programming is described in [Kirchner and Ringeissen, 1998] and [Castro, 1998].

The hybrid language CLAIRE integrates rules and search into an imperative (resp. object-oriented) language [Caseau et al., 2002]. Its use of logic rules enables declarative programming of the type useful for constraint propagation.

In this context, we also mention [Apt, 1998] in which an account of constraint programming from a proof-theoretic perspective is given. Two classes of proof rules are distinguished, 'deterministic' rules formalising constraint propagation, and 'splitting' rules, which correspond precisely to the procedures Propagate and Split, resp., of the general CSP solution algorithm in Fig. 2.1. In this proof-theoretic view, solving a CSP is regarded as proving it from its solutions.

Concurrent constraint programming (CCP) situates the interaction and synchronisation of agents in constraint logic programming [Saraswat, 1993]. Agents use *Ask* and *Tell* operations to publish and query partial information in the form of constraints on shared variables. The constraints are managed in the *constraint store*, which is a set of constraints. AKL [Carlson et al., 1995] and subsequently the Oz language [Smolka, 1995] and the associated Mozart system embody this approach.

Another realisation of the CCP paradigm is the ***Constraint Handling Rules*** (CHR) language. CHR is a declarative high-level language specifically designed for rule-based constraint programming [Frühwirth, 1998]. It is implemented as a language extension that is compiled to the underlying host language; implementations exist in different systems, including the Prolog-based SICStus [M. Carlsson et al., 2004] and Java [Abdennadher et al., 2002]. CHR uses a committed-choice, forward-chaining approach and is intended for constraint propagation. It relies on the host language to provide the search mechanism needed for full constraint solving. In the logic programming approach to constraint programming, CHR is the language of choice to write constraint solvers.

As CHR is closest to our view of rule-based constraint programming, we present it in some detail. CHR supports two principal types of rules:

propagation rules	$H_1, \ldots, H_k \ \Rightarrow\ G_1, \ldots, G_l \ \mid\ B_1, \ldots, B_m,$
and simplification rules	$H_1, \ldots, H_k \ \Leftrightarrow\ G_1, \ldots, G_l \ \mid\ B_1, \ldots, B_m.$

All atomic rule elements can be viewed as constraints, but a distinction is made between defined and primitive constraints:

- the atoms $H_1, \ldots, H_k$ ($k \geqslant 1$) of the rule *head* are defined constraints,
- the atoms $G_1, \ldots, G_l$ ($l \geqslant 0$) of the rule *guard* are built-in constraints,
- the atoms $B_1, \ldots, B_m$ ($m \geqslant 1$) of the rule *body* are arbitrary constraints.

Built-in constraints are provided by the host language (and can also be procedure calls). Defined constraints are managed by the `CHR` runtime system in the `CHR` constraint store. Their definition is, in fact, given by the rules.

A `CHR` rule is executed by first matching its head atoms against constraints in the constraint store. If a match is found, the guard atoms are tested. In case of success, the body atoms are imposed as constraints. If the rule is a simplification rule, additionally the head atoms are removed from the `CHR` constraint store. This process is repeated until no rule matches with successful guards.

The availability of simplification rules makes `CHR` very expressive. Propagation rules just add implied constraints to the constraint store, while simplification rules facilitate non-monotonic updating, so the constraint store can be freely managed. This, and the additionally available host language, makes `CHR` very suitable for high-level design and prototyping of constraint propagation algorithms.

An important issue entailed by the non-monotonicity of simplification rules is that the user must pay attention to confluence and termination of the induced rewriting system.

2.2.2 Rule-Based Constraint Propagation

We now formally introduce our notion of constraint propagation rule.

2.2.1. Definition. Assume that $\mathcal{A}$ and $\mathcal{B}$ are sequences of constraints such that the constraints in $\mathcal{A}$ and $\mathcal{B}$ are on the variables X with domains $\mathcal{D}$. The expression

$$\mathcal{A} \ \rightarrow \ \mathcal{B}$$

is a ***constraint propagation rule***. We call $\mathcal{A}$ the ***condition*** and $\mathcal{B}$ the ***body*** of the rule. Rules act as functions on CSPs. The application of a rule to a CSP with the variables X is given by

$$(\mathcal{A} \rightarrow \mathcal{B})(\ \langle \mathcal{C}, X, \mathcal{D} \rangle\ ) := \begin{cases} \langle \mathcal{C} \cup \mathcal{B}, X, \mathcal{D} \rangle & \text{if } \mathcal{A} \subseteq \mathcal{C}, \\ \langle \mathcal{C}, X, \mathcal{D} \rangle & \text{otherwise.} \end{cases}$$

The rule $\mathcal{A} \rightarrow \mathcal{B}$ is ***correct*** if

$$Sol(\langle \mathcal{A}, X, \mathcal{D} \rangle) \quad \subseteq \quad Sol(\langle \mathcal{B}, X, \mathcal{D} \rangle).$$

In words, a solution of $\mathcal{A}$ is always one of $\mathcal{B}$ as well. □

We capture constraint propagation by such rules. It is easy to verify that the application of a correct rule to a CSP yields an equivalent CSP. The aim is to find *useful* rules, that is, those whose body makes the solutions of a CSP more explicit.

2.2.2. EXAMPLE.

$$x < y,\ y < z \ \rightarrow\ x < z$$

is a constraint propagation rule. Since all solutions of $\langle\{x < y, y < z\}, \langle x, y, z\rangle, \mathcal{D}\rangle$ are solutions of $\langle\{x < z\}, \langle x, y, z\rangle, \mathcal{D}\rangle$ as well, the rule is correct. □

Local Consistency by Rules

The result of constraint propagation is typically characterised by the established local consistency. We now take the inverse view and give simple rule-based characterisations of some local consistency notions, notably generalised arc-consistency and bounds-consistency.

Let us from now on understand $y \neq a$ with the variable y and the constant a as a unary constraint and equally as the domain reduction operation $D_y := D_y \setminus \{a\}$. That is, we assume that ***node consistency*** is maintained. Node consistency [Mackworth, 1977] is the local consistency requiring for a unary constraint $\langle C_R, x\rangle$ with $x \in D_x$ that $C_R = D_x$, whereas generally we only have $C_R \subseteq D_x$.

2.2.3. FACT. Suppose C is a constraint on $X = x_1, \ldots, x_n$.

- Generalised arc-consistency (Def. 2.1.5) on the constraint C is established if all correct rules of the form

  $$C(X) \ \rightarrow\ x_i \neq e$$

  are applied exhaustively. A rule of this form is correct exactly if $e \notin C[x_i]$.

- Bounds-consistency (Def. 2.1.8) on the constraint C is established if all correct rules of the form

  $$C(X) \ \rightarrow\ x_i \neq e \qquad \text{where} \qquad e \in \{\min(D_{x_i}), \max(D_{x_i})\}$$

  are applied exhaustively. A rule of this form is correct exactly if $e \notin C[x_i]$.

  Alternatively we can formulate that bounds-consistency on C is established if all correct rules of the form

  $$\begin{aligned} C(X) \ &\rightarrow\ x_i < e \qquad \text{or} \\ C(X) \ &\rightarrow\ x_i > e \end{aligned}$$

  are applied exhaustively.

□

It is not difficult to see how these characterisations follow directly from the respective definitions.

It is instructive to point out here how constraint propagation can be viewed in terms of the *constraint language* used in the rules: GAC is obtained by stating all variable-value disequalities, while BC is enforced by variable-value inequalities. [Maher, 2002b] studies this topic in detail.

Membership Rules

A specific class of constraint propagation rules are the ***membership rules***, introduced in [Apt and Monfroy, 2001]. These propagation rules have the form

$$C(x_1, \ldots, x_n, y_1, \ldots, y_m),\ x_1 \in S_1, \ldots, x_n \in S_n \ \rightarrow\ y_1 \neq a_1, \ldots, y_m \neq a_m,$$

where each S_i is a set of constants, and each a_i is a constant. An expression $x_i \in S_i$ is a unary constraint on x_i, but in the presence of node consistency it can also be viewed as the simple check $D_i \subseteq S_i$ on the current domain of x_i. We require $S_i \neq \varnothing$ for all $i \in [1..n]$. We also assume that the variables $x_1, \ldots, x_n, y_1, \ldots, y_m$ are pairwise distinct. We call C the ***constraint associated with*** the rule.

In the following, the constraint associated with a rule is usually clear from the context or irrelevant; we then omit it from the notation. If an S_i is equal to the variable base domain, then the (always satisfied) condition $x_i \in S_i$ is often omitted as well. When each set S_i in a membership rule is a singleton set, we call the rule an ***equality rule***.

2.2.4. Example. Consider the constraint $C = \{(0,0),(0,1),(1,1)\}$ on the variables x, y with the base domain $D = \{0,1\}$. The rules

$$y \in \{0\} \ \rightarrow\ x \neq 1,$$
$$x \in \{1\} \ \rightarrow\ y \neq 0,$$

associated with C, are correct. □

The relevance of membership rules for constraint satisfaction problems with finite domains stems from the following observations [Apt and Monfroy, 2001]:

- constraint propagation can be achieved naturally by repeated application of membership rules;
- in particular, the notion of generalised arc-consistency can be characterised in terms of membership rules;
- for constraints explicitly defined on small finite domains, all correct membership rules can be automatically generated;
- many rules used in specific constraint solvers written in the `CHR` (Constraint Handling Rules) language are in fact membership rules.

2.2.5. Example. Reconsider the constraint and the rules of Example 2.2.4. The two rules establish GAC on their associated constraint. □

Membership rules and the template of a GAC-enforcing rule stated in Fact 2.2.3 are clearly connected. Namely, the correctness condition in Fact 2.2.3 is *inserted into the condition* of the rule. Moreover, the form of this correctness condition changes from a test on the constraint into tests on the variable domains. This has practical benefits since the constraint definition is irrelevant.

Chapter 3

Rule Schedulers

3.1 Introduction

In the rule-based approach to constraint programming, the computation process is limited to a repeated application of the propagation rules intertwined with splitting (labelling). The viability of this approach crucially depends on the availability of efficient schedulers for such rules. This motivates the work reported here. We provide an abstract framework for such schedulers and instantiate it for a case of constraint propagation rules, the membership rules. This leads to an implementation that yields a considerably better performance for these rules than the execution of their standard representation as rules in CHR [Frühwirth, 1998].

More precisely, we study schedulers for a generic class of rules which we call *prop* rules. Our approach is explained by the fact that constraint propagation rules, and hence membership rules, are instances of this class. To obtain appropriate schedulers for the *prop* rules we use the generic approach to constraint propagation algorithms introduced in [Apt, 1999] and [Apt, 2000]. In this framework one proceeds in two steps. First, a generic iteration algorithm on partial orderings is introduced and proved correct in an abstract setting. Then it is instantiated with specific partial orderings and functions to obtain specific constraint propagation algorithms. In this chapter, as in [Apt, 2000], we take into account information about the scheduled functions, which are here the *prop* rules. This yields a specific scheduler in the form of an algorithm called R.

We then show by means of an implementation how this abstract framework can be used to obtain a scheduler for membership rules. The implementation is provided as a program in ECL^{i}PS^e [Wallace et al., 1997] that accepts a set of membership rules as input and constructs an ECL^{i}PS^{e}program that is the instantiation of the R algorithm for this set of rules. Since membership rules can be naturally represented as CHR propagation rules, one can assess this implementation by comparing it with the performance of the standard implementation of membership rules in the CHR language. By means of several benchmarks we found that our implementation is considerably faster than `CHR`.

It is important to stress that this implementation is obtained by starting from "first principles" in the form of a generic iteration algorithm on an arbitrary partial ordering. This shows the practical benefits of studying the constraint propagation process on an abstract level.

3.1.1. EXAMPLE. To see the kind of information we use, consider the membership rule

$$x \in \{3, 4, 8\}, y \in \{1, 2\} \ \rightarrow \ z \neq 2.$$

Recall that, informally, it should be read as follows: if the domain of x is included in $\{3, 4, 8\}$ and the domain of y is included in {1,2}, then 2 is removed from the domain of z.

In the computations of constraint programs, the variable domains gradually shrink. Thus, if the domain of x is included in $\{3, 4, 8\}$, then it will remain so during the computation. In turn, if 2 is removed from the domain of z, then this removal operation does not need to be repeated. The concept of a *prop* rule generalises these observations to conditions on the rule premise and body. □

Constraint Handling Rules. The runtime system of the `CHR` language provides a rule scheduler. To make `CHR` usable, it is important that its implementation does not incur too much overhead; and indeed, a great deal of effort was spent on implementing `CHR` efficiently. For an account of the most recent implementation see [Holzbaur et al., 2001]. Since, as already mentioned, many `CHR` rules are membership rules, our approach provides a better implementation of a subset of `CHR`. This, hopefully, may lead to new insights into a design and implementation of languages appropriate for writing constraint solvers.

An important novelty in our approach is the expanded, 'semantic' preprocessing phase during which we analyse the mutual dependencies between the rules. This allows us to remove permanently some rules during the iteration process. This permanent removal of the scheduled rules is particularly beneficial in the context of constraint programming where it leads to accumulated savings when constraint propagation is intertwined with splitting.

3.2 Generic Iteration Algorithm

We begin by recalling the generic algorithm of [Apt, 2000]. We slightly adjust the presentation to our purposes by assuming that the considered partial ordering also has a greatest element $\top$. So we consider a partial ordering $(D, \sqsubseteq)$ with least element $\bot$ and greatest element $\top$, and a set of functions $F = \{f_1, \ldots, f_k\}$ on D. We are interested in functions that satisfy the following two properties.

$\boxed{\mathsf{GI}}$: function set $F \longmapsto$ least common fixpoint

```
d := ⊥
G := F
while  G ≠ ∅  and  d ≠ ⊤  do
        choose g ∈ G
        G := G − {g}
        G := G ∪ update(G, g, d)
        d := g(d)
end
return  d
```

Figure 3.1: Generic Iteration Algorithm GI

3.2.1. Definition.

- f is called *inflationary* if $x \sqsubseteq f(x)$ for all x.
- f is called *monotonic* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for all x, y.

□

The GI algorithm in Fig. 3.1 is used to compute the least common fixpoint of the functions from F. We assume that for all G, g, d the set of functions $update(G, g, d)$ from F is such that

$$\begin{aligned} &\mathbf{A}. \quad \{ f \in F - G \mid f(d) = d \wedge f(g(d)) \neq g(d) \} \subseteq update(G, g, d), \\ &\mathbf{B}. \quad g(d) = d \text{ implies that } update(G, g, d) = \varnothing, \\ &\mathbf{C}. \quad g(g(d)) \neq g(d) \text{ implies that } g \in update(G, g, d). \end{aligned} \tag{3.1}$$

Intuitively, assumption **A** states that $update(G, g, d)$ contains at least all the functions from $F - G$ for which the "old value", d, is a fixpoint but the "new value", $g(d)$, is not. So at each loop iteration such functions are added to the set G. In turn, assumption **B** states that no functions are added to G in case the value of d did not change. Assumption **C** provides information when g is to be added back to G as this information is not provided by **A**. On the whole, the idea is to keep in G at least those functions f for which the current value of d is not a fixpoint.

The use of the condition $d \neq \top$, absent in the original presentation [Apt, 2000], allows us to leave the **while** loop earlier.

Our interest in the GI algorithm is clarified by the following result.

3.2.2. Theorem (Correctness). *Suppose that all functions in F are inflationary and monotonic and that $(D, \sqsubseteq)$ is finite and has least element $\perp$ and greatest element $\top$. Then every execution of the* GI *algorithm terminates and computes in d the least common fixpoint of the functions from F.*

PROOF. Consider the predicate I defined by:

$$\forall f \in F - G.\, f(d) = d \quad \wedge \quad \forall f \in F.\, f(\top) = \top. \tag{I}$$

Note that I is established by the assignment $G := F$. Moreover, it is easy to check that predicate I is preserved by each **while** loop iteration, by virtue of the assumptions **A**, **B** and **C**. Thus, I is an invariant of the **while** loop of the algorithm. So upon its termination

$$(G = \varnothing \ \vee \ d = \top) \quad \wedge \quad I$$

holds, which implies

$$\forall f \in F.\, f(d) = d.$$

This means that the algorithm computes in d a common fixpoint of the functions from F. The rest of the proof is the same as in [Apt, 2000]. So the fact that d is the least common fixpoint follows from the assumption that all functions are monotonic.

Termination is established by considering the lexicographic ordering of the strict partial orderings $(D, \sqsupset)$ and $(\mathbb{N}, <)$, defined on the elements of $D \times \mathbb{N}$ by

$$(d_1, n_1) <_{lex} (d_2, n_2) \quad \text{if} \quad d_1 \sqsupset d_2 \quad \text{or} \quad (d_1 = d_2 \quad \text{and} \quad n_1 < n_2).$$

With each **while** loop iteration of the algorithm, the pair $(d, |G|)$ strictly decreases in the well-founded ordering $<_{lex}$. □

3.3 Revised Generic Iteration Algorithm

We now revise the GI algorithm by modifying dynamically the set of functions that are being scheduled. The idea is that, whenever possible, we remove functions from the set F. This will allow us to exit the loop earlier and can also simplify the **update** operations, which speeds up the execution of the algorithm.

To this end, we assume that for each function $g \in F$ and each element $d \in D$, two sequences[1] of functions from F are given, *friends*(g, d) and *obviated*(g, d), to be instantiated below. We modify the GI algorithm in such a way that each application of g to d will be immediately followed by the applications of all functions from *friends*(g, d) and by a removal of the functions from *friends*(g, d) and from *obviated*(g, d) from F and G. The modified algorithm is called RGI; see Fig. 3.2.

We now formalise the condition that ensures correctness of this scheduling strategy, that is, under which the Correctness Theorem 3.2.2 holds with the GI

[1] We need in it sequences instead of sets since the considered functions will be applied in a specific order. For simplicity, we regard these sequences as sets in some places.

$\boxed{\mathsf{RGI}}$: set of functions $F \longmapsto$ their least common fixpoint

$d := \bot$
$G := F$
while $G \neq \varnothing$ **and** $d \neq \top$ **do**
  choose $g \in G$
  $G := G - \{g\}$
  let $Del = \mathit{friends}(g, d) \cup \mathit{obviated}(g, d)$
  let $h = g \circ g_1 \circ \cdots \circ g_k$ where $\mathit{friends}(g, d) = \langle g_1, \ldots, g_k \rangle$
  $F := F - Del$
  $G := G - Del$
  $G := G \cup \mathit{update}(G, h, d)$
  $d := h(d)$
end
return d

Figure 3.2: Revised Generic Iteration Algorithm RGI

algorithm replaced by the RGI algorithm. Informally, this condition states that after an application of all the functions from $\mathit{friends}(g, d)$ the functions from $\mathit{friends}(g, d)$ and from $\mathit{obviated}(g, d)$ will never change subsequent values of d. We use the notion of stability.

3.3.1. Definition. Suppose $f \in F$ and $d \in D$.

- We say that f is *stable above* d if $d \sqsubseteq e$ implies $f(e) = e$.
- We say that f is *stable* if it is stable above $f(d)$ for all $d \in D$.

□

That is, f is stable if for all d and e, $f(d) \sqsubseteq e$ implies $f(e) = e$. Hence, stability implies idempotence, which means that $f(f(d)) = f(d)$, for all d. Moreover, if d and $f(d)$ are comparable for all d, then stability also implies inflationarity. Indeed, if $d \sqsubseteq f(d)$, then the claim holds vacuously. If $f(d) \sqsubseteq d$, then by stability $f(d) = d$.

Consider now the condition

$$\begin{aligned} &\forall d.\, \forall e \sqsupseteq h(d).\, \forall f \in \mathit{friends}(g, d) \cup \mathit{obviated}(g, d).\ f(e) = e \\ &\text{where } h = g \circ g_1 \circ \cdots \circ g_k \text{ and } \mathit{friends}(g, d) = \langle g_1, \ldots, g_k \rangle. \end{aligned} \tag{3.2}$$

That is, for all elements d, each function f in $\mathit{friends}(g, d) \cup \mathit{obviated}(g, d)$ is stable above $g \circ g_1 \circ \cdots \circ g_k(d)$. The following result holds.

3.3.2. THEOREM. *Suppose that all functions in F are inflationary and monotonic and that $(D, \sqsubseteq)$ is finite and has the least element $\bot$ and the greatest element $\top$. Additionally, suppose that for each function $g \in F$ and $d \in D$ two sequences of functions from F are given, friends(g, d) and obviated(g, d), such that condition* (3.2) *holds.*

Then the Correctness Theorem 3.2.2 holds with the GI *algorithm replaced by the* RGI *algorithm.*

PROOF. Denote by F_0 the initial value of F. In view of condition (3.2), the following statement is an invariant of the **while** loop:

$$\begin{aligned} &\forall f \in F - G.\, f(d) = d \quad \wedge \\ &\forall f \in F.\, f(\top) = \top \quad \wedge \\ &\forall f \in F_0 - F.\, \forall e \sqsupseteq d.\, f(e) = e. \end{aligned} \tag{3.3}$$

Hence, upon termination of the algorithm, the conjunction of this invariant with the negation of the loop condition, i. e.,

$$G = \varnothing \ \vee \ d = \top$$

holds, which implies that $\forall f \in F_0.\, f(d) = d$. The rest of the proof is the same. □

3.4 Functions in the Form of Rules

In what follows we consider the situation when the scheduled functions are of a specific form $b \rightarrow g$, where b is a *condition* and g a function, which we call a *body*. We call such functions *rules*.

First, we explain how rules are applied. Given an element d of D, a condition b is evaluated in d. The outcome is either *true*, which we denote by *holds*(b, d), or *false*. Given a rule $b \rightarrow g$ we define its application to d by

$$(b \rightarrow g)(d) \ = \ \begin{cases} g(d) & \text{if } holds(b, d), \\ d & \text{otherwise.} \end{cases}$$

We are interested in a specific type of conditions and rules.

3.4.1. DEFINITION. Consider a partial ordering $(D, \sqsubseteq)$.

- We say that a condition b is *monotonic* if for all d, e we have that, if $d \sqsubseteq e$ then *holds*(b, d) implies *holds*(b, e).
- We say that a condition b is *precise* if the least d exists such that *holds*(b, d). We call then d the *witness* for b.
- We call a rule $b \rightarrow g$ a *prop* rule if b is monotonic and precise and g is stable.

□

3.4.1 Rules over Sets

To see how natural this class of rules is consider the following case. Take a set A and consider the partial ordering

$$(\mathcal{P}(A), \subseteq).$$

In this ordering the empty set $\varnothing$ is the least element and A is the greatest element. We consider rules of the form

$$B \ \rightarrow \ G,$$

where $B, G \subseteq A$.

To clarify how they are applied to subsets of A we first stipulate for $E \subseteq A$ that

$$holds(B, E) \quad \text{if} \quad B \subseteq E.$$

Then we view a set G as a function on $\mathcal{P}(A)$ by stipulating

$$G(E) \ = \ G \cup E.$$

This determines the application of $B \rightarrow G$.

It is straightforward to see that such rules are *prop* rules. In particular, the element B of $\mathcal{P}(A)$ is the witness for the condition B. For the stability of G, take $E \subseteq A$ and suppose $G(E) \subseteq F$. Then $G(E) = G \cup E$, so $G \cup E \subseteq F$, which implies $G \cup F = F$, i. e., $G(F) = F$.

Such rules can be instantiated to many situations. For example, we can view the elements of the set A as constraints and obtain constraint propagation rules in this way. Alternatively, we can view A as a set of some atomic formulas and each rule $B \rightarrow G$ as a proof rule, usually written as

$$\frac{B}{G}.$$

The minor difference with the usual proof-theoretic framework is that rules have then a single conclusion. An axiom is a rule with the empty set $\varnothing$ as the condition. A closure under such a set of rules is the set of all (atomic) theorems that can be proved using them.

The algorithm presented below can in particular be used to compute efficiently the closure under such proof rules given a finite set of atomic formulas A.

3.4.2 The R Algorithm

We now modify the RGI algorithm for the case of *prop* rules. In the R algorithm in Fig. 3.3 below, we take into account that an application of a rule is a two

$\boxed{\mathsf{R}}$: set of rules $F \longmapsto$ their least common fixpoint

$d := \bot$
$G := F$
while $G \neq \varnothing$ **and** $d \neq \top$ **do**
  choose $(b \to g) \in G$
  $G := G - \{b \to g\}$
  if $\mathit{holds}(b, d)$ **then**
    let $\mathit{Del} = \mathit{friends}(b \to g) \cup \mathit{obviated}(b \to g)$
    let $h = g \circ g_1 \circ \cdots \circ g_k$
      where $\mathit{friends}(b \to g) = \langle (b_1 \to g_1), \ldots, (b_k \to g_k) \rangle$
    $F := F - \mathit{Del}$
    $G := G - \mathit{Del}$
    $G := G \cup \mathit{update}(G, h, d)$
    $d := h(d)$
  else if $\forall e \sqsupseteq d.\ \neg \mathit{holds}(b, e)$ **then**
    $F := F - \{b \to g\}$
  end
end
return d

Figure 3.3: Rule scheduling algorithm R

step process: testing of the condition followed by a conditional application of the body. This allows us to drop the parameter d from the sequences $\mathit{friends}(g, d)$ and $\mathit{obviated}(g, d)$ and consequently to construct such sequences before the execution of the algorithm. The sequence $\mathit{friends}(g)$ will be constructed in such a way that we shall not need to evaluate the conditions of its rules: they will all hold. The specific construction of the sequences $\mathit{friends}(g)$ and $\mathit{obviated}(g)$ is provided in a second algorithm below, the *Friends and Obviated Algorithm*.

Again, we are interested in identifying conditions under which the Correctness Theorem 3.2.2 holds with the GI algorithm replaced by the R algorithm. To this end, given a rule $b \to g$ in F and $d \in D$, define as follows:

$$\mathit{friends}(b \to g, d) = \begin{cases} \mathit{friends}(b \to g) & \text{if } \mathit{holds}(b, d), \\ \langle\rangle & \text{otherwise,} \end{cases} \tag{3.4}$$

and

$$\mathit{obviated}(b \to g, d) = \begin{cases} \mathit{obviated}(b \to g) & \text{if } \mathit{holds}(b, d), \\ \langle b \to g \rangle & \text{if } \forall e \sqsupseteq d\ \neg \mathit{holds}(b, e), \\ \langle\rangle & \text{otherwise.} \end{cases} \tag{3.5}$$

We obtain the following counterpart of the Correctness Theorem 3.2.2.

3.4.2. THEOREM (CORRECTNESS). *Suppose that all functions in F are prop rules of the form $b \to g$, where g is inflationary and monotonic, and that $(D, \sqsubseteq)$ is finite and has the least element $\bot$ and the greatest element $\top$. Further, assume that for each rule $b \to g$ the sequences friends$(b \to g, d)$ and obviated$(b \to g, d)$, defined as above, satisfy condition* (3.2) *and the following condition:*

$$\forall d. \big(b' \to g' \in \mathit{friends}(b \to g) \wedge \mathit{holds}(b, d) \;\to\; \forall e \sqsupseteq g(d).\, \mathit{holds}(b', e)\big). \quad (3.6)$$

Then the Correctness Theorem 3.2.2 holds with the GI *algorithm replaced by the* R *algorithm.*

PROOF. It suffices to show that the R algorithm is an instance of the RGI algorithm. On account of condition (3.6) and the fact that the rule bodies are inflationary functions, $\mathit{holds}(b, d)$ implies that

$$\big((b \to g) \circ (b_1 \to g_1) \circ \cdots \circ (b_k \to g_k)\big)\,(d) \;=\; (g \circ g_1 \circ \cdots \circ g_k)(d),$$

where $\mathit{friends}(b \to g) = \langle (b_1 \to g_1), \ldots, (b_k \to g_k) \rangle$. This takes care of the situation in which $\mathit{holds}(b, d)$ is true.

In turn, the definition of $\mathit{friends}(b \to g, d)$ and $\mathit{obviated}(b \to g, d)$ and assumption **B** take care of the situation when $\neg \mathit{holds}(b, d)$. When the condition b fails for all $e \sqsupseteq d$ we can conclude that for all such e we have $(b \to g)(e) = e$. This allows us to remove at that point of the execution the rule $b \to g$ from the set F. This amounts to adding $b \to g$ to the set $\mathit{obviated}(b \to g, d)$ at runtime. Note that condition (3.2) is then satisfied. □

We now provide an explicit construction of the sequences *friends* and *obviated* for a rule $b \to g$ in the form of the $\mathsf{F\,\&\,O}$ algorithm in Fig. 3.4. $\mathsf{GI}(F, e)$ stands there for the GI algorithm invoked with $\bot$ replaced by e. We call a rule r *relevant in an execution of* $\mathsf{GI}(F, e)$ if at some point in this execution $r(d) \neq d$ holds after the "choose $r \in G$" action.

The $\mathsf{F\,\&\,O}$ algorithm needs the witness of the rule condition b, that is, the least d for which $\mathit{holds}(b, d)$. For the rules we are interested in most, the witness can be easily extracted from the condition; see Section 3.4.1 for rules over sets and Section 3.6.2 for membership rules.

Note that the rule $r = b \to g$ itself is not in $\mathit{friends}(r)$ as it is a *prop* rule. It is contained in $\mathit{obviated}(r)$, however, since $g(d) = d$ holds by the stability of g. In Section 3.6.2, we give a concrete example for the sequences *friends*, *obviated* using membership rules.

The following observation shows the adequacy of the $\mathsf{F\,\&\,O}$ algorithm for our purposes.

$\boxed{\mathsf{F\&O}}$: rule r in rule set $F \longmapsto \langle \mathit{friends}(r), \mathit{obviated}(r) \rangle$

let $r = b \to g$
let w be the witness of b

$d := \mathsf{GI}(F, g(w))$
$\mathit{friends}$:= sequence of the relevant rules $h \in F$ in the preceding execution of GI

$\mathit{obviated} := \langle\rangle$
for each $(b' \to g') \in F - \mathit{friends}$ **do**
  if $g'(d) = d$ **or** $\forall e \sqsupseteq d.\, \neg \mathit{holds}(b', e)$ **then**
    $\mathit{obviated} := (b' \to g'), \mathit{obviated}$
  end
end
return $\langle \mathit{friends}, \mathit{obviated} \rangle$

Figure 3.4: Friends and Obviated Algorithm $\mathsf{F\,\&\,O}$

3.4.3. NOTE. *Upon termination of the* $\mathsf{F\,\&\,O}$ *algorithm, conditions* (3.2) *and* (3.6) *hold, where the sequences* $\mathit{friends}(b \to g, d)$ *and* $\mathit{obviated}(b \to g, d)$ *are defined as in Equations* (3.4) *and* (3.5). □

Let us summarise the findings of this section that culminated in the R algorithm. Assume that all functions are in the form of rules satisfying the conditions of the Correctness Theorem 3.4.2. Then in the R algorithm, each time the evaluation of the condition b of the selected rule $b \to g$ succeeds,

- the rules in the sequence $\mathit{friends}(b \to g)$ are applied directly without testing the value of their conditions,
- the rules in $\mathit{friends}(b \to g) \cup \mathit{obviated}(b \to g)$ are permanently removed from the currently active set of functions G and from F.

3.5 Recomputing Least Fixpoints

Another substantial benefit of the sequences $\mathit{friends}(b \to g)$ and $\mathit{obviated}(b \to g)$ surfaces when the R algorithm is repeatedly applied to compute the least fixpoint. More specifically, consider the following sequence of actions:

- we compute the least common fixpoint d of the functions from F,
- we move from d to an element e such that $d \sqsubseteq e$,
- we compute the least common fixpoint above e of the functions from F.

Such a sequence of actions typically arises in the framework of CSPs, further studied in Section 3.6. There, the computation of the least common fixpoint d of the functions from F corresponds to the constraint propagation process for a constraint C. The move from d to e such that $d \sqsubseteq e$ corresponds to splitting or constraint propagation involving another constraint, and the computation of the least common fixpoint above e of the functions from F corresponds to a subsequent round of constraint propagation for C.

Suppose now that we computed the least common fixpoint d of the functions from F using the RGI algorithm or its modification R for the rules. During its execution we permanently removed some functions from the set F. These functions are then not needed for computing the least common fixpoint above e of the functions from F. The precise statement is provided in the following simple, yet crucial, theorem.

3.5.1. THEOREM. *Suppose that all functions in F are inflationary and monotonic and that $(D, \sqsubseteq)$ is finite. Suppose that the least common fixpoint d_0 of the functions from F is computed by means of the* RGI *or* R *algorithm. Let F_{fin} be the final value of the variable F upon termination of the* RGI *or* R *algorithm.*

Suppose now that $d_0 \sqsubseteq e$. Then the least common fixpoint e_0 above e of the functions from F coincides with the least common fixpoint above e of the functions from F_{fin}.

PROOF. Take a common fixpoint e_1 of the functions from F_{fin} such that $e \sqsubseteq e_1$. It suffices to prove that e_1 is a common fixpoint of the functions from F.

So take $f \in F - F_{fin}$. Since condition (3.3) is an invariant of the main **while** loop of the RGI algorithm and of the R algorithm, it holds upon termination, and consequently f is stable above d_0. But $d_0 \sqsubseteq e$ and $e \sqsubseteq e_1$, so we conclude that $f(e_1) = e_1$. □

Intuitively, this result means that, if after splitting we relaunch the same constraint propagation process, we can disregard the removed functions. We illustrate this important effect with a concrete example in Section 3.7.4.

3.6 Concrete Framework

We now proceed with our main goal, namely an instantiation of the scheduler algorithm framework for the case of membership rules. We have indicated in Section 3.4.1 a possible instantiation of the *prop* rule framework to constraint propagation rules, of which membership rules are a special case. We set up a different instantiation for these rules now, however. This specialised instantiation is more natural as it is based on domains, and membership rules essentially deal only with domains.

3.6.1 Partial Orderings

With each CSP $\mathcal{P} = \langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ we associate now a specific partial ordering. Initially we take the Cartesian product of the partial orderings $(\mathcal{P}(D_1), \supseteq), \ldots, (\mathcal{P}(D_n), \supseteq)$. So this ordering is of the form

$$(\mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n), \supseteq)$$

where we interpret $\supseteq$ as the Cartesian product of the reversed subset ordering. The elements of this partial ordering are sequences $(E_1, \ldots, E_n)$ of respective subsets of $(D_1, \ldots, D_n)$ ordered by the component-wise reversed subset ordering. Note that $(D_1, \ldots, D_n)$ is the least element in this ordering while

$$\underbrace{(\varnothing, \ldots, \varnothing)}_{n \text{ times}}$$

is the greatest element. However, we would like to identify with the greatest element all sequences that contain the empty set as an element. So we divide the above partial ordering by an equivalence relation $R_\varnothing$ according to which

$$\begin{array}{lll} (E_1, \ldots, E_n)\ R_\varnothing\ (F_1, \ldots, F_n) & \text{if} & (E_1, \ldots, E_n) = (F_1, \ldots, F_n) \\ & & \text{or} \\ & & \exists i.\, E_i = \varnothing \quad \text{and} \quad \exists j.\, F_j = \varnothing. \end{array}$$

It is straightforward to see that $R_\varnothing$ is indeed an equivalence relation. In the resulting quotient ordering there are two types of elements:

- the sequences $(E_1, \ldots, E_n)$ that do not contain the empty set as an element, and which we continue to present in the usual way with the understanding that now each of the listed sets is non-empty;
- one special element equal to the equivalence class consisting of all sequences that contain the empty set as an element. This equivalence class is the greatest element in the resulting ordering, so we denote it by $\top$.

In what follows we denote this partial ordering by $(D_\mathcal{P}, \sqsubseteq)$.

3.6.2 Membership Rules

Fix a specific CSP $\mathcal{P} = \langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ with finite domains. Let C be one of its constraints on the variables $y_1, \ldots, y_k, z_1, \ldots, z_m$. We recall the notion of membership rule from Section 2.2.2. The rule

$$C,\ y_1 \in S_1, \ldots, y_k \in S_k\ \rightarrow\ z_1 \neq a_1, \ldots, z_m \neq a_m$$

is a membership rule associated with C. $a_1, \ldots, a_m$ are constants, and $S_1, \ldots, S_k$ are constant subsets of the respective variable domains, We drop here the condition that the sequences $y_1, \ldots, y_k$ and $z_1, \ldots, z_m$ have no variable in common so that we can combine membership rules.

Let us reformulate the interpretation of such rules so as to fit the framework considered in the previous section. To this end, we need to clarify how to

- evaluate the condition of a membership rule in an element of the considered partial ordering,
- interpret the conclusion of a membership rule as a function on the considered partial ordering.

Let us start with the first item.

3.6.1. Definition. Given a variable y with the domain D_y and $E \in \mathcal{P}(D_y)$ we define

$$holds(y \in S, E) \quad \text{if} \quad E \subseteq S,$$

and extend the definition to the elements of the ordering $(D_{\mathcal{P}}, \sqsubseteq)$ by putting

$$holds(y \in S, (E_1, \ldots, E_n)) \quad \text{if} \quad E_k \subseteq S, \quad \text{where we assumed that } y \text{ is } x_k,$$

and

$$holds(y \in S, \top).$$

Furthermore we interpret a sequence of conditions as a conjunction, by putting

$$holds((y_1 \in S_1, \ldots, y_k \in S_k),\ (E_1, \ldots, E_n)) \quad \text{if} \quad holds(y_i \in S_i,\ (E_1, \ldots, E_n)) \quad \text{for } i \in [1..k].$$

□

It is not difficult to see what the witness of a membership rule condition is. Consider the CSP $\langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ and its associated partial ordering. The witness of $y_1 \in S_1, \ldots, y_k \in S_k$ is $(E_1, \ldots, E_n)$ where $E_i = S_k$ if $x_i = y_k$, and $E_i = D_k$ if x_i does not occur in the condition.

Concerning the second item we proceed as follows.

3.6.2. Definition. Given a variable z with the domain D_z we interpret the atomic formula $z \neq a$ as a function on $\mathcal{P}(D_z)$, defined by:

$$(z \neq a)(E) = E - \{a\}.$$

Then we extend this function to the elements of the considered ordering $(D_{\mathcal{P}}, \sqsubseteq)$ as follows:

- on the elements of the form $(E_1, \ldots, E_n)$ we put

  $$(z \neq a)(E_1, \ldots, E_n) = (E'_1, \ldots, E'_n),$$

  where

  $$E'_i = \begin{cases} E_i - \{a\} & \text{if } z \equiv x_i, \\ E_i & \text{otherwise.} \end{cases}$$

  If the resulting sequence $(E'_1, \ldots, E'_n)$ contains the empty set, then we replace it by $\top$,

- on the element $\top$ we put $(z \neq a)(\top) = \top$.

Finally, we interpret a sequence $z_1 \neq a_1, \ldots, z_m \neq a_m$ of atomic formulas by interpreting each of them in turn. □

3.6.3. Example. Take the CSP $\langle \mathcal{C};\ x_1, \ldots, x_4 \in \{a, b, c\}\rangle$ and consider the membership rule

$$x_1 \in \{a, b\},\ x_2 \in \{b\} \quad \rightarrow \quad x_3 \neq a,\ x_3 \neq b,\ x_4 \neq a. \tag{r}$$

Then we have

$$\begin{aligned} r(\ \{a\}, \{b\}, \{a,b,c\}, \{a,b\}\ ) &= (\{a\}, \{b\}, \{c\}, \{b\}), \\ r(\ \{a,b,c\}, \{b\}, \{a,b,c\}, \{a,b\}\ ) &= (\{a,b,c\}, \{b\}, \{a,b,c\}, \{a,b\}), \\ r(\ \{a,b\}, \{b\}, \{a,b\}, \{a,b\}\ ) &= \top. \end{aligned}$$

The witness of r is $(\{a,b\}, \{b\}, \{a,b,c\}, \{a,b,c\})$. □

In view of the Correctness Theorem 3.4.2, the following observation allows us to apply the R algorithm when each function is a membership rule and when for each rule $b \rightarrow g$ the sequences $\mathit{friends}(b \rightarrow g)$ and $\mathit{obviated}(b \rightarrow g)$ are constructed by the F & O algorithm.

3.6.4. Note. *Consider the partial ordering $(D_{\mathcal{P}}, \sqsubseteq)$.*

1. *Each membership rule is a prop rule.*
2. *Each function $z_1 \neq a_1, \ldots, z_m \neq a_m$ on $D_{\mathcal{P}}$ is*
   - *inflationary,*
   - *monotonic.*

□

To be able to instantiate the R algorithm with the membership rules, we still need to define the set $update(G, g, d)$. In our implementation we chose the following simple definition:

$$update(G, g, d) = \begin{cases} F - G & \text{if } g(d) \neq d, \\ \varnothing & \text{otherwise.} \end{cases}$$

Note that assumptions **A**, **B**, **C** in (3.1) hold.

3.6.5. Example. Let us illustrate the intuition behind the use of the sequences $friends(b \to g)$ and $obviated(b \to g)$. Take again $\langle \mathcal{C};\ x_1, \ldots, x_4 \in \{a, b, c\}\rangle$ and consider the membership rules

$$\begin{aligned} x_1 \in \{a, b\} \quad &\to \quad x_2 \neq a, x_4 \neq b, && (r_1) \\ x_1 \in \{a, b\},\ x_2 \in \{b, c\} \quad &\to \quad x_3 \neq a, && (r_2) \\ x_2 \in \{b\} \quad &\to \quad x_3 \neq a, x_4 \neq b, && (r_3) \\ x_2 \in \{a\} \quad &\to \quad x_1 \neq a. && (r_4) \end{aligned}$$

Upon application of rule r_1, rule r_2 can be applied without evaluating its condition. Subsequently, rule r_3 can be deleted without applying it since its body has become irrelevant; the same holds for r_1 itself. Finally, rule r_4 can be deleted since its condition can now never succeed. Hence, we can have

$$\begin{aligned} friends(r_1) &= \langle r_2 \rangle, \qquad \text{and} \\ obviated(r_1) &= \langle r_1, r_3, r_4 \rangle, \end{aligned}$$

which is in fact what the F & O algorithm computes. □

3.7 Implementation

In this section we discuss the implementation of the R algorithm for the membership rules and compare it by means of several benchmarks with the CHR implementation in the ECL^{i}PS^e system.

3.7.1 Modelling Membership Rules in CHR

Following [Apt and Monfroy, 2001], membership rules are represented as CHR propagation rules with a single head. Recall from Section 2.2.1 that the latter ones are of the form

$$H \ \Rightarrow\ G_1, \ldots, G_l \ \mid\ B_1, \ldots, B_m$$

where the atom H of the rule head is a defined constraint, the atoms $G_1, \ldots, G_l$ of the rule guard are built-in constraints, and the atoms $B_1, \ldots, B_m$ of the rule body are arbitrary constraints.

Let us also review how CHR propagation rules with one head are executed. First, given a query (that represents a CSP) the variables of the rule are renamed to avoid variable clashes. Then an attempt is made to match the head of the rule against the first atom of the query. If it is successful and all guards of the instantiated version of the rule succeed, the instantiated version of the body of the rule is executed. Otherwise the next rule is tried.

Finally, let us recall the representation of a membership rule as a CHR propagation rule from [Apt and Monfroy, 2001]. We assume that the host language is ECL^{i}PS^e. Consider the membership rule

$$y_1 \in S_1, \ldots, y_k \in S_k \;\rightarrow\; z_1 \neq a_1, \ldots, z_m \neq a_m$$

associated with the defined constraint `c` on the variables $\mathtt{X_1}, \ldots, \mathtt{X_n}$. We represent its condition by starting initially with the atom $\mathtt{c}(\mathtt{X_1}, \ldots, \mathtt{X_n})$ as the head. Each atomic condition of the form $y_i \in \{a\}$ is processed by replacing in the atom $\mathtt{c}(\mathtt{X_1}, \ldots, \mathtt{X_n})$ the variable y_i by a. In turn, each atomic condition of the form $y_i \in S_i$, where S_i is not a singleton, is processed by adding the atom $\mathtt{in}(\mathtt{y_i}, \mathtt{LS_i})$ to the guard of the propagation rule. The `in/2` predicate is defined by

```
in(X, L) :- dom(X, D), subset(D, L).
```

So `in(X,L)` holds if the current domain of the variable `X` (yielded by the built-in `dom` of ECL^{i}PS^e) is included in the list `L`. In turn, $\mathtt{LS_i}$ is a list representation of the set S_i.

Finally, each atomic conclusion $z_i \neq a_i$ translates to the atom z_i `##` a_i of the body of the propagation rule.

As an example consider the membership rule

$$X \in \{0\}, Y \in \{1, 2\} \;\rightarrow\; Z \neq 2$$

associated with a constraint c on the variables X, Y, Z. It is represented by the following CHR propagation rule:

```
c(0,Y,Z) ==> in(Y, [1,2]) | Z ## 2.
```

In ECL^{i}PS^e, variables with singleton domains are automatically instantiated. So, assuming that the variable domains are non-empty, the condition of this membership rule holds if and only if the head of the renamed version of the above propagation rule matches the atom `c(0,Y,Z)` *and* the current domain of the variable `Y` is included in `[1,2]`. Further, in both cases the execution of the body leads to the removal of the value 2 from the domain of `Z`. So the execution of both rules has the same effect when the variable domains are non-empty.

Execution of CHR

In general, the application of a membership rule as defined in Section 3.6 and the execution of its representation as a CHR propagation rules coincide. Moreover, by the semantics of CHR, the CHR rules are repeatedly applied until a fixpoint is reached. So a repeated application of a finite set of membership rules coincides with the execution of the CHR program formed by the representations of these membership rules as propagation rules. An important point concerning the standard execution of a CHR program is that, in contrast to the R algorithm, every change in the variable domains of a constraint causes the computation to restart.

3.7.2 Benchmarks

In our approach, the repeated application of a finite set of membership rules is realised by means of the R algorithm of Section 3.3 implemented in ECL^{i}PS^e. The compiler consists of about 1500 lines of code. It accepts as input a set of membership rules, each represented as a CHR propagation rule, and constructs an ECL^{i}PS^e program that is the instantiation of the R algorithm for this set of rules. As in CHR, for each constraint the set of rules that refer to it is scheduled separately.

In the benchmarks below, we used for each considered CSP the sets of all subsumption-free valid membership and equality rules for the 'base' constraints. These rule sets were automatically generated using a program discussed in [Apt and Monfroy, 2001]. In the first phase, the compiler constructs for each rule g the sequences *friends*(g) and *obviated*(g). Time spent on this construction is comparable with the time needed for the generation of the equality and membership rules for a given constraint. For example, the medium-sized membership rule set for the rcc8 constraint, containing 912 rules, was generated in 166 seconds while the construction of all *friends* and *obviated* sequences took 142 seconds time. It is important to note that generating the rules and the sequences *friends*, *obviated* takes place once, at compile-time, while the resulting constraint propagation procedure is typically used many times; hence fast generation is not a critical issue.

To see the impact of the accumulated savings obtained by permanent removal of the rules during the iteration process, we chose benchmarks that embody several successive propagation steps, i. e., propagation interleaved with search (domain splitting or labelling).

In Table 3.1, we list the results for selected single constraints. For each constraint, say C on the variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, we consider the CSP $\langle C; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ together with randomised labelling; i. e., the choices of variable, value, and action (assigning or removing the value), are all random. The computation of simply one or all solutions yields insignificant times, so the benchmark program computes and records also all intermediate

Constr.	rcc8	fork	and3	and9	and11
MEM					
rel.	37% / 22%	58% / 46%	66% / 49%	26% / 15%	57% / 25%
abs.	147/396/686	0.36/0.62/0.78	0.27/0.41/0.55	449/1727/2940	1874/3321/7615
EQU					
rel.	97% / 100%	98% / 94%	92% / 59%	95% / 100%	96% / 101%
abs.	359/368/359	21.6/21.9/22.9	0.36/0.39/0.61	386/407/385	342/355/338

Table 3.1: Randomised search trees for single constraints

Logic	3-valued	9-valued	11-valued
MEMBERSHIP			
relative	61% / 44%	65% / 29%	73% / 29%
absolute	1.37/2.23/3.09	111/172/385	713/982/2495
EQUALITY			
relative	63% / 29%	40% / 57%	36% / 51%
absolute	0.77/1.22/2.70	2.56/6.39/4.50	13.8/38.7/26.7

Table 3.2: CSPs formalising sequential ATPG

non-solution fixpoints. Backtracking occurs if a recorded fixpoint is encountered again. In essence, all possible search trees are traversed. In some cases, this takes too much time; we then limit the number of visited nodes.

In Table 3.2, we list the results for practically motivated CSPs. We chose here CSPs that formalise the problem of automatic test pattern generation for sequential digital circuits (ATPG), to be discussed in Chapter 6. These are large CSPs that employ the and$N$ constraints of Table 3.1 and a number of other constraints, almost all of which are implemented by rules. The constraint and$N(x, y, z)$ expresses the conjunction $x \wedge y = z$ in an N-valued logic.

We measured the execution times for three rule schedulers: the standard CHR representation of the rules, the generic chaotic iteration algorithm GI, and its improved derivative R. The codes of both the latter two algorithms are produced by our compiler and are thus structurally very similar, which allows a direct assessment of the improvements embodied in R.

In the tables, we provide for each constraint or CSP the ratio of the execution times between, first, R and GI, and second, R and CHR. This is followed by the absolute times in seconds in the order R / GI / CHR.

The platform for all benchmarks was a Sun Enterprise 450 with four UltraSPARC-II 400 MHz processors and 2 GB memory under Solaris, and ECL^{i}PS^e 5.5 (in single processor mode).

We find a substantial speedup in many cases when using R, both comparing R and GI, and R and CHR.

Possibilities for Improving the Implementation

We examined some of the various ways of optimising our implementation of the R algorithm in ECL^{i}PS^e. In particular, we considered a better embedding into the constraint-handling mechanism of ECL^{i}PS^e, for instance by finer control of the waking conditions and a joint removal of the elements from the same variable domain instead of several disequality constraints resulting from larger sequences *friends*. Using such techniques, we succeeded in achieving an additional average speed-up by a factor of 4.

This open-ended work indicates that further improvements are possible. For example, an unrealised improvement with a plausible gain in efficiency is a better choice of the data structures for handling the rule sets F and G. We use lists (plain lists and Prolog difference lists), in which, e. g., element finding has linear cost, while in a balanced tree this cost is only logarithmic.

3.7.3 Detecting When a Constraint is Solved

An important point in the implementations is the question of when to remove solved constraints from the constraint store. The standard CHR representation of membership rules as generated by the algorithm of [Apt and Monfroy, 2001] does so by containing, beside the propagation rules, one CHR simplification rule for each tuple in the constraint definition. Once its variables are assigned values that correspond to a solution, the constraint is solved, and removed from the store by the corresponding simplification rule. This 'solved' test takes place interleaved with executing the propagation rules.

The implementations of GI and R, on the other hand, check *after* closure under the propagation rules. The constraint is considered solved if all its variables are fixed (necessarily to a solution), or, in the case of R, if the set F of remaining rules is empty; this is discussed in the following subsection. Interestingly, comparing CHR and GI, the additional simplification rules sometimes constitute a substantial overhead while at other times their presence allows earlier termination.

3.7.4 Recomputing Least Fixpoints

Let us finally illustrate the impact of the permanent removal of rules during the least fixpoint computation, achieved here by the use of the sequences *friends*(r) and *obviated*(r).

3.7.1. DEFINITION. Given a set F of rules, we call a rule $g \in F$ *solving* if $\mathit{friends}(g) \cup \mathit{obviated}(g) = F$. □

Take as an example the ternary equivalence relation $\equiv$ from the three-valued logic of [Kleene, 1952, p. 334] that consists of the values, 0 (true), 1 (false) and u (unknown). For instance, we have $\equiv(1, \mathsf{u}, \mathsf{u})$. The full definition is given by the following truth table:

$\equiv$	1	0	u
1	1	0	u
0	0	1	u
u	u	u	u

The program of [Apt and Monfroy, 2001] generates 26 minimal valid membership rules for the $\equiv$ constraint. Out of them, 12 are solving rules. For the remaining rules the sizes of the set $\mathit{friends} \cup \mathit{obviated}$ are: 17 (for 8 rules), 14 (for 4 rules), and 6 (for 2 rules).

In the R algorithm, a selection of a solving rule leads directly to termination, $G = \varnothing$, and to a reduction of the set F to $\varnothing$. For other rules, a considerable simplification in the computation takes place. For example,

$$x \in \{0\}, z \in \{0, \mathsf{u}\} \;\rightarrow\; y \neq 0 \tag{r}$$

is one of the 8 rules of which the set $\mathit{friends}(r) \cup \mathit{obviated}(r)$ has size 17.

Consider now the CSP

$$\langle \equiv (x, y, z)\,;\, x \in \{0\}, y \in \{0, 1, \mathsf{u}\}, z \in \{0, \mathsf{u}\}\rangle\,.$$

In the R algorithm, the selection of r is followed by the application of the rules from $\mathit{friends}(r)$ and the removal of the rules from $\mathit{friends}(r) \cup \mathit{obviated}(r)$. This brings the number of the considered rules down to $26 - 17 = 9$. The R algorithm subsequently discovers that none of these rules is applicable at this point. So the nine rules remain upon termination.

In a subsequent constraint propagation phase, launched after splitting or after constraint propagation involving another constraint, the fixpoint computation by means of the R algorithm involves only these nine rules instead of the initial 26!

For solving rules, this fixpoint computation terminates immediately.

Solving Rules

Interestingly, as Table 3.3 shows, solving rules occur quite frequently for equality rules, bot less often so for non-equality membership rules. We list for each constraint and each type of rules the number of solving rules divided (/) by the total number of rules, followed in a new line by the average number of rules in

Constraints	`and2`	`and3`	`and9`	`and11`	`fork`	`rcc8`	`allen`
EQUALITY	6/6	13/16	113/134	129/153	9/12	183/183	498/498
	6	14	130	148	11	183	498
MEMBERSHIP	6/6	4/13	72/1294	196/4656	0/24	0/912	n.a./26446
	6	7	810	3156	9	556	n.a.

Table 3.3: Solving rules

the set $friends(r) \cup obviated(r)$. The rule sets were computed using the program of [Apt and Monfroy, 2001].

The fork constraint is taken from the Waltz language for the analysis of polyhedral scenes. The rcc8 constraint represents the composition table of the Region Connection Calculus with 8 relations from [Egenhofer, 1991] (which we revisit in Chapter 9). It is remarkable that all its 183 equality rules are solving. While none of the 912 membership rule for rcc8 is solving, on average the set $friends(r) \cup obviated(r)$ contains 556 membership rules. Also all 498 equality rules for the allen constraint, which represents the composition table of Allen's thirteen qualitative temporal relations [Allen, 1983], are solving. The number of membership rules exceeds 26 000 and consequently they are too costly to analyse.

`CHR` Simplification Rules

The `CHR` language supports besides propagation rules also so-called simplification rules. Using such rules, one can remove constraints from the constraint store, so one can freely affect its form. In [Abdennadher and Rigotti, 2001], a method is discussed that allows one to automatically transform `CHR` propagation rules into simplification rules such that the semantics of the rule set is respected. The method is based on identifying or constructing propagation rules that are solving.

In contrast, our method captures the *degree* to which a rule is solving. We define

$$\text{solving degree of } r \in \mathcal{R} \quad = \quad \frac{|friends(r) \cup obviated(r)|}{|\mathcal{R}|}.$$

If this ratio is 1 then r is a solving rule. More typically, the ratio will be less than 1. Consider Figure 3.5 for the distribution of the solving degree of the rules for `and9`. Only 72 rules have degree 1 and correspond thus to simplification rules.

Let *Del* abbreviate $friends(r) \cup obviated(r)$. Consider now two non-solving rules r_1, r_2, that is, such that $Del(r_1) \neq \mathcal{R}$ and $Del(r_2) \neq \mathcal{R}$. But let also $Del(r_1) \cup Del(r_2) = \mathcal{R}$. Suppose that during a fixpoint computation the conditions of both rules have succeeded, and their bodies have been applied. The R algorithm detects immediately that the constraint is solved, and terminates

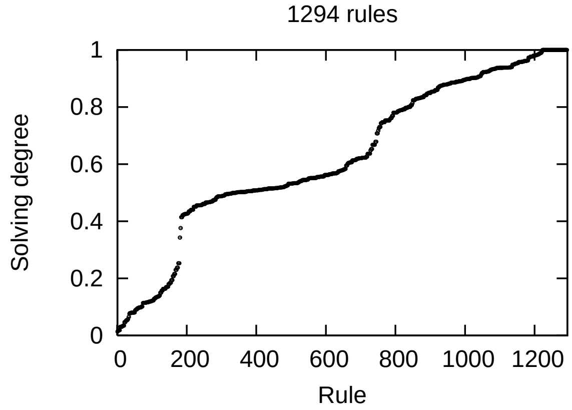

Figure 3.5: Membership rules for `and9`: Solving degree

consequently. `CHR`, for which r_1 and r_2 are ordinary propagation rules, cannot detect this possibility for immediate termination. In R, we observe an *accumulated effect* of removing rules from the fixpoint computation.

We revisit the issue of the relevance of solving rules for the R scheduler in Section 4.6 of the following chapter.

3.8 Final Remarks

We studied the problem of efficient scheduling of constraint propagation rules. Starting from a generic iteration algorithm for functions, we obtained the R scheduler by step-wise refinement. The central observation exploited in the R algorithm is that an application of some constraint propagation rule in which its condition succeeds may provide the justification to immediately apply other rules without testing their condition, or to remove other rules from the iteration. Removing a rule in the R algorithm is ultimate in the sense that a removed rule need not be reconsidered in subsequent propagation rounds, which can therefore be expected to be faster. This is important, as a constraint propagation algorithm is typically executed repeatedly, interleaved both with other propagation and search.

We described an implementation of the R scheduler for membership rules, and we gave experimental evidence for the value of the efficiency improvements by comparing the R scheduler with a generic iteration algorithm and a `CHR` implementation by way of benchmarks. We found substantial speedups in many cases. Finally, we argued that the increase in efficiency is not due to implicitly distinguishing solving rules from non-solving rules, but by accumulating the effect of removing rules.

Chapter 4

Redundant Rules

4.1 Introduction

Given a set of constraint propagation rules, a natural question is whether each rule is needed for the desired constraint propagation. It may be that the effect of applying some rule r can also be obtained by applying one or several other rules. In this case, removing rule r from the rule set does not affect the result of constraint propagation associated with the rule set.

4.1.1. EXAMPLE. Consider the set $R = \{r_1, \ldots, r_4\}$ of constraint propagation rules, given as follows:

$$\begin{array}{rcl} a, b & \rightarrow & c, \\ b & \rightarrow & c, \\ c & \rightarrow & d, \\ b & \rightarrow & d. \end{array} \qquad \begin{array}{r} (r_1) \\ (r_2) \\ (r_3) \\ (r_4) \end{array}$$

Rule r_1 is unneeded in presence of rule r_2. Indeed, whenever r_1 can add the constraint c then also r_2 can. Nor is rule r_4 needed: its effect can always be obtained by applying two rules, r_2 followed by r_3.

Hence, the rule set $\{r_2, r_3\}$ propagates as much as R. □

Constraint propagation rules are employed in fixpoint computation algorithms. An ideal algorithm would schedule the rules in such a way that the induced derivation becomes shortest. Practical algorithms, such as GI and its derivatives studied in the previous chapter, try to keep derivations short, but generally the cost of a fixpoint computation rises with the number of rules involved. This explains the interest in small rule sets. One way to obtain small sets is to identify rules that are unneeded for computing the common fixpoints.

We examine here the issue of redundancy with respect to fixpoint computation for sets of functions that are in the form of rules. Specifically, we deal with *prop* rules, introduced in the previous chapter; see Definition 3.4.1. The concept of redundancy is formalised in a "semantic" sense that takes into account the type of computations performed by means of the considered rules. We provide a simple test for redundancy that leads to a natural way of computing minimal rules sets in an appropriate sense.

Redundancy in rule-based programs in the `CHR` language is examined in [Abdennadher and Frühwirth, 2002]. Since `CHR` is very expressive, the proposed redundancy test is necessarily quite abstract, relying on termination, confluence, and operational equivalence of original and reduced program. The test is also computationally more expensive than our test for the case of *prop* rules.

The issue of identifying redundant rules is highly relevant for the *automatic generation* of constraint propagation rules. Two significant such methods are described in [Apt and Monfroy, 2001] and [Abdennadher and Rigotti, 2004]. Both approaches employ notions of redundancy and avoid generating such rules. However, these redundancy notions are not general enough or only informally defined. We show that they are subsumed by our comprehensive and rigourous approach. According to our notion, the mentioned rule generation methods may produce rules that are (in part) unneeded for computing common fixpoints of the respective rule sets.

To show relevance and feasibility of our approach, we discuss an ECL^{i}PS^e implementation of the computation of minimal rule sets by redundancy removal. We report the outcome of applying the minimisation technique to several sets of specific constraint propagation rules stemming from the rule generation methods mentioned above, and we assess by benchmarks the effect that using the smaller rule sets has on propagation performance.

4.2 Redundant Functions

We start again with arbitrary functions before moving on to *prop* rules. In the following, for brevity, we drop the word "common" when referring to common fixpoints of a set of functions.

4.2.1. DEFINITION.

- Consider a set $F \cup \{f\}$ of functions on a partial ordering. A function f is called *redundant with respect to* F if the sets of fixpoints of F and $F \cup \{f\}$ are equal.
- A set of functions F is called *minimal with respect to redundancy* (or simply *minimal*) if no function $f \in F$ is redundant with respect to $F - \{f\}$.

$\square$

Equivalently, we can say that a function f is redundant w. r. t. F if every fixpoint of F is also a fixpoint of f.

4.3 Redundant Rules

We focus now on the subject of redundancy of *prop* rules, and formulate the following simple criterion.

4.3.1. THEOREM. *Consider a set F of prop rules and a prop rule $r = (b \to g)$ with the witness w for b. Let e be the least fixpoint of F greater than or equal to w. If and only if $g(e) = e$, then the rule r is redundant with respect to F.*

PROOF. We show first that $g(e) = e$ implies that an arbitrary fixpoint d of F is also a fixpoint of r. We make a case distinction on the condition.

b **holds for** d**:** So $r(d) = g(d)$. We have $w \sqsubseteq d$ since w is the witness for b. Also, $w \sqsubseteq e \sqsubseteq d$ since e is the least fixpoint of F greater than or equal to w. From $e \sqsubseteq d$, $g(e) = e$, and the stability of g we conclude $g(d) = d$. Hence $r(d) = g(d) = d$.

b **does not hold for** d**:** Then $r(d) = (b \to g)(d) = d$.

The "only if" part is proved by showing that $g(e) \neq e$ implies that F and $F \cup \{r\}$ have different fixpoints. This is the case: consider e. □

This test is of interest to us since it requires to compute only one fixpoint of F instead of all fixpoints. It is effective if

- the witness can be computed,
- the equality $g(e) = e$ can be determined, and
- the fixpoint computations are effective.

Partial Redundancy

For the sake of fixpoint computations, a rule $r = (b \to g)$ with the body $g = g_1, \ldots, g_n$ describing the function composition $g_1 \circ \cdots \circ g_n$, such that any two different functions g_i, g_j commute, can be identified with the collection $(b \to g_1), \ldots, (b \to g_n)$ of rules, and vice versa. Indeed, the respective fixpoints and the rule properties are maintained. We consider here these two representations as largely equivalent.

If a rule with such a 'compound' body is not redundant then it might still be so in part. That is, some part of its body might be redundant or, in other words, some sub-rules of its decomposition might be. In that case we say that the rule is *partially* redundant.

We argue in Section 4.5.2 below that eliminating partial redundancy improves the performance of fixpoint computations with the R algorithm, introduced in Section 3.4.2.

$\boxed{\mathsf{MinRuleSet}}$: rule set $F \longmapsto$ a corresponding minimal rule set

for each $r \in F$, in some order **do**
  if $\mathsf{Redundant}(r, F - \{r\})$ **then** $F := F - \{r\}$
end

$\boxed{\mathsf{Redundant}}$: rule $b \to g$, rule set $F \longmapsto$ true/false

$w :=$ witness for b
$e := \mathsf{CommonFixpoint}(F, w)$
if $g(e) = e$ **then return** true **else return** false

Figure 4.1: Rule set minimisation

4.3.1 Computing Minimal Sets of *prop* Rules

Rule set minimisation can be achieved by a simple bounded loop (Fig. 4.1). It is important to observe that *several* minimal rule sets correspond to a given non-minimal set in general. The obtained minimal set depends on the selection order for testing (see Example 4.3.3 further down).

A reasonable strategy is to test first those rules that are undesirable, hoping that they are redundant and thus expendable. The criterion in our implementation processing constraint propagation rules is that a rule is comparatively undesirable if its condition is expensive to test (because it consists of many constraints), and its body is weakly constraining (because it consists of few constraints). We also apply minimisation in two phases: first, only fully redundant rules are eliminated, then, every partially redundant rule is reduced. In this way, we hope to obtain a set of rules for which fixpoint computations are generally fast.

4.3.2 Subsumption

We highlight a common special case of redundancy, involving only two rules. Informally, a rule subsumes another if its condition is at least as weak and its body is at least as strong. For example, $c_1 \to c_3, c_4$ subsumes $c_1, c_2 \to c_3$. We adopt the term 'subsumption' from automated reasoning where it denotes a similar concept.

4.3.2. Corollary. *Consider a set F of prop rules and two rules $r_1 = (b_1 \to g_1)$ and $r_2 = (b_2 \to g_2)$ such that $r_1 \in F$ and $r_2 \notin F$. Assume that g_2 is inflationary and that, for all d,*

$$holds(b_2, d) \quad implies \quad holds(b_1, d), \qquad and \qquad g_2(d) \sqsubseteq g_1(d).$$

Then the rule r_2 is redundant with respect to F.

$$
\begin{array}{llll}
c(x,y,z,0) & \rightarrow & x \neq 0, y \neq 0, z \neq 0 & (1)\\
c(x,y,1,u) & \rightarrow & u \neq 1, \underline{x \neq 0}, \underline{y \neq 0} & (2)\\
c(0,y,z,u) & \rightarrow & u \neq 0, y \neq 0, \underline{z \neq 1} & (3)\\
c(x,0,z,u) & \rightarrow & u \neq 0, x \neq 0, \underline{z \neq 1} & (4)\\
c(x,y,z,1) & \rightarrow & z \neq 1 & (5)\\
c(x,y,0,u) & \rightarrow & u \neq 0 & (6)\\
c(1,1,z,u) & \rightarrow & u \neq 1, \underline{z \neq 0} & (7)\\
c(x,1,0,u) & \rightarrow & x \neq 1 & (8)\\
c(x,1,z,1) & \rightarrow & \underline{x \neq 1} & (9)\\
c(1,y,0,u) & \rightarrow & y \neq 1 & (10)\\
c(1,y,z,1) & \rightarrow & \underline{y \neq 1} & (11)
\end{array}
$$

Figure 4.2: Membership rules for the constraint c

PROOF. Let e be the least fixpoint of F greater than or equal to the witness w_2 of b_2. We show that $g_2(e) = e$, which entails the desired result by Theorem 4.3.1.

We have $holds(b_2, w)$, so by monotonicity of b_2 also $holds(b_2, e)$. The first requirement above implies $holds(b_1, e)$. We know for the fixpoint e that $e = r_1(e)$, and with $holds(b_1, e)$ also $e = g_1(e)$. By the second requirement we conclude $g_2(e) \sqsubseteq e = g_1(e)$, but g_2 is also inflationary: $e \sqsubseteq g_2(e)$. Hence, $g_2(e) = e$. □

4.3.3. EXAMPLE. Let us illustrate a number of issues with respect to redundant rules by means of an example. Consider the constraint $c(x, y, z, u)$ defined by

x	y	z	u
0	1	0	1
1	0	0	1
1	1	1	0

The underlying domain for all its variables is $\{0, 1\}$. The induced corresponding partial order is

$$(\{\, (A, B, C, D) \mid A, B, C, D \subseteq \{0, 1\}\,\}, \supseteq),$$

following the formalisation in Section 3.6.1. The rule generation algorithm of [Apt and Monfroy, 2001] generates eleven membership rules, listed in Fig. 4.2 (since the rule conditions are only equality tests, we use an alternative, compact notation).

Suppose we are interested in computing the smallest fixpoint greater than or equal to $E_1 = \{1\} \times \{0,1\} \times \{0,1\} \times \{1\}$. Suppose rule (11) is considered. Its application yields $E_2 = \{1\} \times \{0\} \times \{0,1\} \times \{1\}$ from where rule (4) leads to $E_3 = \{1\} \times \{0\} \times \{0\} \times \{1\}$. This indeed is a fixpoint since for each rule either its condition does not apply or the application of its body results again in E_3.

A second possible iteration from E_1 that stabilises in E_3 is by rule (5) followed by rule (10). Rule (11) can be applied at this point but its body does not change E_3. Indeed, E_3 is a fixpoint of all rules including rule (11). From the fact that E_1 is the witness of the condition of rule (11), we conclude that rule (11) is redundant — in fact, we just performed the test of Theorem 4.3.1.

The process of identifying redundant rules can then be continued for the rule set $\{(1), \ldots, (10)\}$. One possible outcome is depicted in Figure 4.2, where redundant parts of rule bodies are underlined. 7 out of the total of 20 initial atomic conclusions are deleted, so we find here a redundancy ratio of 35%.

Consider now the justification for the redundancy of rule (11), and observe that rule (11) has no effect since rule (10), which has the same body, was applied before. Suppose now that the process of redundancy identification is started with rule (10) instead of rule (11). This strategy results in rule (10) being identified as redundant, with a relevant application of rule (11).

Note moreover that one of the rules (10), (11) must be present in *any* minimal set since their common body $y \neq 1$ occurs in no other rule. This suggests that sometimes several equally useful minimal sets exist that correspond to a given non-minimal set.

4.4 Implementation and Empirical Evaluation

We implemented in ECL^{i}PS^e the MinRuleSet algorithm in two instantiations, one for a specific class of automatically generated constraint propagation rules and one for membership rules.

4.4.1 Constraint Propagation Rules

Constraint propagation rules with conditions and bodies consisting of various multiple constraints can be automatically generated using the RuleMiner algorithm of [Abdennadher and Rigotti, 2004].

In RuleMiner, several criteria are used to identify an undesired rule. The single most important one is called *lhs-cover*. A rule $C_1 \to C_2$ is called lhs-covered by $C_3 \to C_4$ if $C_1 \supseteq C_3$ and $C_2 \subseteq C_4$, where the C_i are sets of constraints. This requirement is implied by the condition of Corollary 4.3.2, which can be seen if we abstract constraint propagation rules to *prop* rules as in Section 3.4.1. The notion of lhs-covering is a special case of subsumption and, in turn, general redundancy.

	and	or	xor	andor	andxor	orxor	andor+	andxor+	orxor+
total	19	19	28	138	207	199	176	254	246
redundant									
total	–	–	–	–	–	–	–	–	–
partial	7	7	1	83	82	77	135	192	184
redundancy ratio	24%	24%	3%	38%	21%	21%	61%	54%	54%

Table 4.1: Redundancy in RuleMiner rule sets

The authors of the RuleMiner algorithm [Abdennadher and Rigotti, 2004] kindly provided us with several generated rule sets for the constraints $\mathsf{and}, \mathsf{or}, \mathsf{xor}$, which correspond to the logical operators in a 6-valued logic. The rules are used in the automatic generation of test patterns for digital circuits, an electrical engineering problem which we discuss in Chapter 6. For the semantics of the 6-valued logic, see specifically Section 6.2.1. The constraint $\mathsf{and}(x, y, z)$ captures $x \wedge y = z$ in the corresponding logic.

The given RuleMiner rules capture propagation from single constraints and pairs of constraints. In both cases, additional atomic equality constraints between two variables, or a variable and a constant, may occur in a rule condition. The body of a rule consists of equality and disequality constraints.

Here are two example rules, using the original compact notation:

$$\mathsf{and}(x, x, z) \quad \rightarrow \quad x \neq \overline{\mathsf{d}}, x \neq \mathsf{d}, x = z, \tag{1}$$

$$\mathsf{and}(x, y, z), \mathsf{or}(z, y, 1) \quad \rightarrow \quad z \neq \overline{\mathsf{d}}, z \neq \mathsf{d}, x = z, y = 1. \tag{2}$$

The rules can be rewritten so as to fit the format of abstract propagation rules, by introducing new variables and equalities in the rule conditions. For example,

$$\mathsf{and}(x, y, z), \mathsf{or}(z, y, 1) \qquad \text{is} \qquad \mathsf{and}(x, y, z), \mathsf{or}(u, v, w), z = u, y = v, w = 1.$$

We assume appropriate rules for equality constraints, i. e., expressing transitivity and symmetry. These rules are considered part of the rule set to be minimised but are excluded from being tested for redundancy themselves.

The results for some test rule sets are in Table 4.1. We provide the size of the original rule set, the number of redundant and partially redundant rules, and the redundancy ratio, which is the percentage of atomic constraints that were removed from rule bodies.

The first three columns in Table 4.1 describe the results for rule sets corresponding to the single logical constraints, that is, rules such as (1). The three centre columns contain the results for rule sets for pairs of logical constraints,

	and	or	xor	andor	andxor	orxor	andor+	andxor+	orxor+
total	19	19	28	138	207	199	176	254	246
redundant									
total	–	–	–	6	–	–	18	12	12
partial	7	7	1	77	86	81	117	180	172
redundancy ratio	24%	24%	3%	39%	22%	21%	63%	55%	55%

Table 4.2: Redundancy in RuleMiner rule sets, with domain information

i. e., rules such as (2). Finally, the last three columns correspond to the union of the rule sets for a pair of logical constraints and its respective two individual constraints, i. e., rules as (1) and (2) together. This configuration corresponds to the intended use.

The tested RuleMiner rule sets contained partial redundancies, but they did not contain any (totally) redundant rules. This observation may surprise since lhs-covering is not a very strong redundancy notion. However, additional redundancy criteria are employed in the RuleMiner system. In particular, an ad-hoc minimisation is conducted. It takes place *during* rule generation: the redundancy of a rule is checked directly after its generation. This means that the subsequently generated rules are not taken into account.

Adding domain information. From a semantical point of view, one piece of information that is not available in our example RuleMiner rules are the variable domains. The central constraints represent logical operators $\mathsf{and}, \mathsf{or}, \ldots$ in a 6-valued logic. Using the rules as intended implies that the constrained variables have the corresponding 6-valued domain; let us call it $D_6 = \{0, 1, \mathsf{d}, \ldots\}$. This means that in this case one can augment the condition of each rule by unary domain constraints $v \in D_6$ for all variables v occurring in the rule. So rule (1) could then be written as

$$\mathsf{and}(x, x, z), x \in D_6, z \in D_6 \quad \rightarrow \quad x \neq \overline{\mathsf{d}}, x \neq \mathsf{d}, x = z. \tag{1$'$}$$

This additional information, which is available to the RuleMiner generator, is relevant for redundancy minimisation as it changes the witness of the rule condition. To see the effect, consider a situation in which some variable v is involved in five disequality constraints with different constants. Then, $v \in D_6$ entails that v is equal to the remaining 6th value.

Table 4.2 reports the rule set sizes and redundancy ratios for the RuleMiner rules augmented with domain information. Some of the rules are redundant.

	$\mathsf{and11}_M$	$\mathsf{and11}_E$	$\mathsf{and3}_M$	$\mathsf{equ3}_M$	$\mathsf{fula2}_E$	fork_E	fork_M
total	4656	153	18	26	52	12	24
redundant							
total	4263	–	5	8	24	–	6
partial	2	6	–	–	–	9	6
redundancy ratio	81%	4%	30%	26%	35%	35%	40%

Table 4.3: Minimising rule sets

4.4.2 Membership Rules

The algorithm described in [Apt and Monfroy, 2001], which we call RGA (and quote in Fig. 5.1 in the following chaper), can be used to generate a set of membership rules from a constraint definition. Its only redundancy concept is that of *extension*. In our notation, the membership rule $r_2 = (b_2 \rightarrow g_2)$ extends the rule $r_1 = (b_1 \rightarrow g_1)$ if $holds(b_2, d)$ implies $holds(b_1, d)$ and $g_1 = g_2$. Rule r_2 extending r_1 is redundant w. r. t. r_1.

The concept of extension is a special case of our notion of subsumption, Corollary 4.3.2. This suggests that the RGA algorithm of [Apt and Monfroy, 2001] may still generate rules that are redundant according to our wider criterion.

This is indeed the case. We applied rule set minimisation according to Theorem 4.3.1 to some generated benchmark membership rule sets. The results are listed in Table 4.3. The constraints are taken from the experiments reported in Table 3.1 of the previous chapter. Additionally, a 5-ary constraint fulladder (abbreviated to fula) is analysed. It captures the addition of two bits with additional input and output carry bits.

For each rule set, it is indicated by the respective subscript M or E whether it was generated as a set of equality rules or a set of membership rules (sufficient to enforce GAC on the constraint). The numeric suffix to logical constraints states the size of the logic.

We observe redundancy in all examined rule sets of Table 4.3. In the case of the ternary $\mathsf{and11}_M$ constraint, which expresses the conjunction $x \wedge y = z$ in an 11-valued logic, minimising the original rule set results in an enormous reduction to just 393 rules. In Section 4.5.2, we report experiments in which the rules are used for propagation (for example, using the minimised rule set for propagating $\mathsf{and11}_M$ speeds up the computations by a factor of 10).

4.5 Discussion

For the complete CHR language, the issue of redundancy is examined in [Abdennadher and Frühwirth, 2002], using an approach based on term rewriting concepts (see, e. g., [Baader and Nipkow, 1998] for an introduction). The class of CHR rule-based programs is strictly more expressive than the class of *prop* rules. The central difference is the presence of simplification rules, which remove constraints from the constraint store. CHR rules are thus generally neither monotonic nor inflationary. Consequently, the proposed redundancy test needs to be more abstract than ours, relying on termination, confluence, and operational equivalence of original and reduced rule sets instead.

For *prop* rules viewed as a term rewriting system, termination and confluence are guaranteed, and Theorem 4.3.1 constitutes a concrete test of operational equivalence. Benefiting from inflationarity and monotonicity, we can do with only one fixpoint computation per candidate rule, whereas, if the rules are viewed as a CHR program, two computations are needed, with and without the candidate.

Completion in Term Rewriting Systems

A link exists between redundancy and the completion of term rewriting systems. Completion adds rules to a rule set so as to make it confluent, that is, to prevent the existence of some point from which two iterations stabilise in different fixpoints. In such a case, a new rule is introduced that joins both iterations, effectively removing one fixpoint. So the new rule enables an alternative iteration that leads to the same remaining fixpoint.

Redundancy removal, in contrast, tries to *minimise* the number of alternative iterations leading to the same fixpoint, while maintaining the total set fixpoints. This is done by removing a rule that occurs in one possible iteration but not in all of them.

4.5.1 Benefit of Rule Set Minimisation

It is difficult to argue *generally* that minimising rule sets is useful when the rule sets are used for computing common fixpoints. While it seems obvious that discarding a larger number of redundant rules accelerates fixpoint computation, this is not so clear when removing one single rule.

A redundant rule can also be viewed as a short-cut, which typically requires several other rules to simulate if removed. For an appropriate choice of scheduling strategy, rule set, and starting point of the fixpoint computation, the effect of redundancy removal on the computation time may consequently be adverse.

This issue is even more relevant for the case of a partially redundant rule. Therefore, we can not state that reducing redundancy is always useful (although in our experiments that was the case). However, observe that partial redundancy can easily be reintroduced.

4.5.2 Minimal Rule Sets and the R Scheduler

The R scheduler, Section 3.4.2, uses sets of rules *friends*(r) and *obviated*(r) for each rule r. After an application of r in which its condition held, the rules in both sets become irrelevant for the remainder of the computation. These rule become 'locally' redundant. No trivial connection between redundancy and the rule sets *friends*(r) and *obviated*(r) exists, however.

4.5.1. NOTE. *Let F be a set of rules used in the* R *scheduler, and abbreviate*

$$Del(r) = friends(r) \cup obviated(r)$$

for each rule $r \in F$.

- *It is* not *the case that a rule is redundant w. r. t. F if it is contained in $Del(r)$ of every rule $r \in F$.*
- *Nor is a redundant rule necessarily contained in $Del(r)$ of every rule $r \in F$.*

□

Here are the counter examples.

4.5.2. EXAMPLE. Recall the rule set $F = \{(1), \ldots, (11)\}$ of Fig. 4.2. We find

$$Del(r) = \begin{cases} \{(1), (2), (5), (6)\} & \text{if } r = (5) \text{ or } r = (6), \\ F & \text{otherwise,} \end{cases}$$

for rules $r \in F$.

Observe that rule (5) is contained in the set $Del(r)$ for all rules r. However, rule (5) is not redundant with respect to F. On the other hand, rule (11) is redundant with respect to F, but it is not contained in each set $Del(r)$. □

Partial Redundancy Removal for the R Scheduler

It is useful to remove partial redundancies when the R scheduler is used. The reason is that the set $Del(r)$ for rules $r \in F$ to be scheduled can sometimes be larger if partially redundant rules are reduced. Note that partial redundancies removed from a rule are not lost but reassociated with it by the set *friends*(r) of the R scheduler. Informally and slightly simplified, *friends*(r) collects those rules whose condition necessarily succeeds after a relevant application of r.

4.5.3. Example. We consider the logical and constraint in the three-valued logic of [Kleene, 1952, p. 334]. The program of [Apt and Monfroy, 2001] generates for it a set of 22 membership rules, which shrinks to a set of 13 rules by removing redundancies. Three rules from the obtained minimal rule set, which we call F, associated with $\mathsf{and}(x, y, z)$ are

$$x \in \{0, \mathsf{u}\} \quad \to \quad z \neq 1 \qquad (r_1)$$

$$y \in \{1, \mathsf{u}\}, z \in \{0\} \quad \to \quad x \neq 1 \qquad (r_2)$$

$$y \in \{1, \mathsf{u}\}, z \in \{0, 1\} \quad \to \quad x \neq \mathsf{u} \qquad (r_3)$$

We can have $r_2 \in \mathit{obviated}(r_1)$, since the body $x \neq 1$ of r_2 is irrelevant once r_2 has fired, which requires $x \in \{0, \mathsf{u}\}$. Furthermore, we may have $r_3 \in \mathit{friends}(r_2)$ since the condition of r_3 is implied by the condition of r_2.

Let us modify r_2 by composing it with r_3. So we redefine r_2 as the partially redundant rule

$$y \in \{1, \mathsf{u}\}, z \in \{0\} \quad \to \quad x \neq 1, x \neq \mathsf{u}.$$

This change does not affect the common fixpoints of F, nor does it make any rule in F fully redundant. It does, however, change the set $\mathit{obviated}(r_1)$, of which r_2 can not be a member now. In the R scheduler, slower convergence results. ☐

Benchmarks

To see what effect the absence of redundancy on the relative performance of the R scheduler has, we reran the benchmarks reported in Tables 3.1 and 3.2 of the previous chapter. All involved rule sets were subjected to a redundancy removal, and subsequently, recomputations of the respective sets $\mathit{friends}(r)$ and $\mathit{obviated}(r)$ for each rule r were performed. The results are shown in Tables 4.4 and 4.5. The rule sets of $\mathsf{rcc8}$ were already minimal; therefore this constraint is omitted.

When comparing the redundancy and non-redundancy benchmark versions, we observe that the absolute execution times are enormously reduced in the case of the constraints on higher-valued logics, by a factor of rougly 10 in the case of $\mathsf{and11}_M$, for example. This is in line with the much smaller sizes of the reduced rule sets. The ratios of the execution times, however, are much less affected. Judging from these observations, the type of scheduler and minimality w. r. t. redundancy appear to be orthogonal issues. Hence, both optimisation opportunities are relevant and should be exploited.

Distribution of the Solving Degree

It is interesting to examine in one case the distribution of the solving degrees, i. e., the ratios of the sizes of $\mathit{friends}(r) \cup \mathit{obviated}(r)$ and the full rule set, for a rule r. Recall that a ratio of 1 means that the constraint is solved once the rule

Constraint	`fork`	`and3`	`and9`	`and11`
MEMBERSHIP				
relative	60% / 46%	69% / 48%	28% / 18%	50% / 29%
absolute	0.32/0.53/0.70	0.27/0.39/0.56	167/589/924	157/316/543
EQUALITY				
relative	97% / 93%	97% / 64%	96% / 101%	96% / 101%
absolute	21.6/22.2/23.2	0.37/0.38/0.58	386/404/384	341/353/339

Table 4.4: Randomised search trees for single constraints (no redundant rules)

Logic	3-valued	9-valued	11-valued
MEMBERSHIP			
relative	66% / 46%	62% / 33%	68% / 35%
absolute	1.32/2.00/3.05	37/59/114	70/103/199
EQUALITY			
relative	61% / 26%	40% / 58%	33% / 48%
absolute	0.72/1.18/2.73	2.57/6.41/4.46	13.8/41.0/28.6

Table 4.5: CSPs formalising sequential ATPG (no redundant rules)

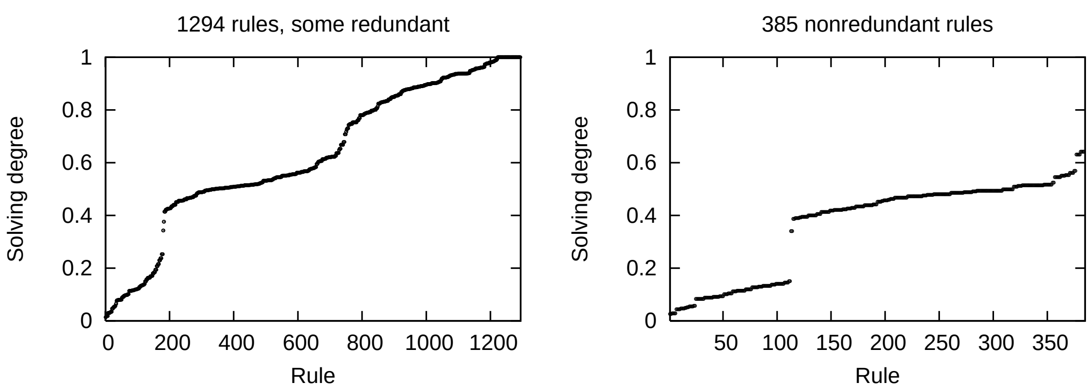

Table 4.6: $\texttt{and9}_M$: Solving degree and redundancy

body has been executed. Such a rule could be represented as a simplification rule in CHR (see Section 3.7.4).

In Table 4.6 two membership rule sets for the constraint and9 are compared. One set contains redundant rules, the other set is minimal w. r. t. redundancy. The rules in the minimal set are solving to a lesser degree; in particular, none is a proper solving rule. The good performance of the R algorithm in the benchmarks of Tables 4.4, 4.5 can thus not be attributed to distinguishing solving (simplification) rules and non-solving propagation rules, but is due to the accumulated effect of removing rules from the fixpoint computation.

4.6 Final Remarks

We studied the issue of redundancy in sets of constraint propagation rules. A rule in a rule set is redundant if removing it from the set does not weaken the propagagation associated with the set. Our redundancy notion is simple, comprehensive, and generalises several notions described, sometimes informally, in the literature. We gave an algorithm to minimise rule sets with respect to redundancy. Redundancy removal is an indispensable technique in the automatic generation of constraint propagation rules. We showed experimentally that several rule generation methods produce redundant rules. Moreover, we demonstrated that removing redundancy can result in substantial speedups when using the rule sets for constraint propagation. Finally, we showed that redundancy removal is orthogonal to the improvements embodied in the R scheduler, which entails that both techniques should be used together.

One open question results from the fact that the rule selection strategy during minimisation generally has an effect on the obtained minimal rule set: what criterion should be used to compare two minimal sets, and what strategy is appropriate to find preferred minimal sets.

Chapter 5

Incremental Rule Generation

5.1 Introduction

While constraint propagation rules capturing the desired propagation of one or some constraints can be devised manually, in doing so, several issues arise. Designing appropriate rules requires expertise; their correctness must be guaranteed, and for more complex constraints, it may not even realistically be possible. In response to these difficulties, the issue of an *automatic generation* of rule-based constraint propagation algorithms has received considerable attention in recent years. [Apt and Monfroy, 2001] considers the generation of membership rules; [Ringeissen and Monfroy, 2000] examines a parameterised variant of them. [Abdennadher and Rigotti, 2002, Abdennadher and Rigotti, 2004] deal with more general constraint propagation rules. The latter approaches aim particularly at the `CHR` language and also discuss methods to generate `CHR` simplification rules allowing the deletion of constraints from the constraint store (see Section 2.2.1). In [Dao et al., 2002], the issue of automatic generation of solvers based on indexicals [Codognet and Diaz, 1996] is examined.

Common to most of these approaches is their paradigm that is essentially generate-and-test. Successively, candidate rules for constraint propagation are enumerated. A rule candidate is kept if it passes the correctness test against the constraint definition. In the deviating method of [Ringeissen and Monfroy, 2000], a conclusion is derived from a candidate premise, which itself comes from a syntactic enumeration process, however. [Abdennadher and Frühwirth, 2003] examines how to merge solvers written in the `CHR` language. Due to the expressiveness of `CHR`, the main aspects are termination and confluence. Here we concern ourselves only with constraint propagation rules where these two properties are no issues, which lets us focus on the constraint propagation.

In contrast to the generate-and-test approaches, we explore the idea of rule generation by incrementally modifying previously constructed rule sets. The key feature is that the input to the solver generation algorithm is already a set of rules.

The generation process consists in transforming the rule set into one that possesses desirable properties with respect to the associated constraints, such as the ability to establish a local consistency. An explicit definition of the constraints, for instance extensionally as the set of solutions, is unnecessary. The rule set is processed according to declarative transformation steps, *meta rules*, leading to the introduction of new rules or removal of existing rules. A number of benefits arise from this approach:

- First, the description of rule generation as an incremental process provides a *new perspective* on the origins of rule-based constraint solvers. This helps us to better understand such solvers and their propagation.
- Second, incremental solver generation *reuses* previously constructed rule sets. It also potentially *increases* the level of constraint propagation.
- Third, the incremental method can also be used as a universal rule generation method, by accompanying it with a pre-process that turns a constraint definition not based on rules into a set of simple initial rules.

While we first discuss incremental rule generation in general, the main part of this chapter deals with a specific type of rule, the membership rules. Our motivation for this focus is, on the one hand, that few useful statements can be made without fixing a specific language of constraint propagation rule (we elaborate on this issue below), and on the other hand, the relevance of membership rules.

We examine a variety of cases of incrementally generating sets of membership rules. In a justified sense, the central case is constructing a rule set $R(C_1 \wedge C_2)$ for the conjunctive constraint $C_1 \wedge C_2$ from the rule sets $R(C_1)$ and $R(C_2)$ of its constituent constraints. The simple union $R(C_1) \cup R(C_2)$ generally does not maximally propagate the conjunction $C_1 \wedge C_2$. Take the following rules, for example.

$$C_1, x \in \{\ 1\ \} \ \rightarrow\ C' \tag{r_1}$$
$$C_2, x \in \{\ 2\ \} \ \rightarrow\ C' \tag{r_2}$$
$$C_1 \wedge C_2, x \in \{1, 2\} \ \rightarrow\ C' \tag{r_3}$$

In presence of the conjunctive constraints $C_1 \wedge C_2$ and $x \in \{1, 2\}$, none of the rules r_1, r_2 of the constituent constraints C_1, C_2 lets us obtain C'. This shows why we would like to derive stronger rules such as r_3.

Furthermore, we discuss the cases of existential and universal quantification. If a constraint C is on a variable x then both $\exists x.C$ and $\forall x.C$ are constraints on the remaining variables of C, and we explain how to construct the rule sets $R(\exists x.C)$ and $R(\forall x.C)$ based on $R(C)$. We also discuss the auxiliary cases of extending the scope of a constraint to a new variable, and of extending the underlying domain by a new element, which means adding certain new solutions to the constraint. A method of obtaining rules for a constraint from its extensional definition, alternatively as a set of solutions or non-solutions, makes incremental membership rule generation as capable as competing membership rule generation methods.

In all instances of membership rule generation, we focus on their most relevant feature, namely the relation to generalised arc-consistency.

5.2 Transforming Sets of Constraint Propagation Rules

A transformation of a rule set is a sequence of atomic steps introducing or removing single rules. We describe the admissible steps by *meta rules* with side conditions, applied to sets of constraint propagation rules. We write

$$\frac{R}{R \cup \{r\}} \quad \text{(introduce)} \qquad\qquad \frac{R}{R \setminus \{r\}} \quad \text{(remove)}$$

where R is a rule set and r is a rule.

We consider two meta rules: subsumption, which deletes a rule, and derivation, which introduces a rule, based on the given rules.

5.2.1 Subsumption

Subsumption is a special case of redundancy of a rule with respect to a rule set, for the purpose of computing common fixpoints; see Section 4.3.2. We restrict ourselves here to a simple case and consider only propagation rules with the same body.

As a meta rule, we have

$$\frac{R \cup \{\mathcal{A} \to \mathcal{C},\ \mathcal{B} \to \mathcal{C}\}}{R \cup \{\mathcal{A} \to \mathcal{C}\}} \qquad \text{if (5.1)} \qquad \textsf{(gen-subsume)}$$

where the constraints in $\mathcal{A}, \mathcal{B}$ are on the variables X with domains $\mathcal{D}$, and the side condition is

$$Sol(\langle \mathcal{A}, X, \mathcal{D} \rangle) \supseteq Sol(\langle \mathcal{B}, X, \mathcal{D} \rangle) \tag{5.1}$$

Recall that $Sol(\mathcal{P})$ is the set of solutions of the CSP $\mathcal{P}$, so (5.1) expresses that $\mathcal{A}$ is *implied* by $\mathcal{B}$.

We say that a rule is subsumed by a set of rules if it is subsumed by some rule in the set. So $x < y \to \mathcal{C}$ is subsumed by $R \cup \{x \leqslant y \to \mathcal{C}\}$.

5.2.2 Derivation

Two rules with identical body give rise to a new rule if the disjunction of their conditions, or something more restrictive, can be expressed in the underlying constraint language. Formally,

$$\frac{R \cup \{\mathcal{A}_1 \to \mathcal{C},\ \mathcal{A}_2 \to \mathcal{C}\}}{R \cup \{\mathcal{A}_1 \to \mathcal{C},\ \mathcal{A}_2 \to \mathcal{C}, \mathcal{B} \to \mathcal{C}\}} \quad \text{if (5.2) and (5.3)} \qquad \textsf{(gen-derive)}$$

where the constraints in $\mathcal{A}_1, \mathcal{A}_2, \mathcal{B}$ are on the variables X with domains $\mathcal{D}$, and the side condition is

$$Sol(\langle \mathcal{B}, X, \mathcal{D}\rangle) \ \subseteq\ Sol(\langle \mathcal{A}_1, X, \mathcal{D}\rangle) \cup Sol(\langle \mathcal{A}_2, X, \mathcal{D}\rangle). \tag{5.2}$$

The idea of this transformation step is to compose the ancestor rules, at best into a descendant that in turn subsumes one or both ancestors. It is not difficult to show that correctness is preserved: if each rule in the original rule set is correct then so is each rule in the obtained rule set. However, note that the respective common fixpoints of the rule sets generally change!

While not needed for preservation of correctness, it is useful to require additionally

$$Sol(\langle \mathcal{B}, X, \mathcal{D}\rangle) \ \not\subseteq\ Sol(\langle \mathcal{A}_i, X, \mathcal{D}\rangle) \qquad \text{for } i = 1 \text{ and } i = 2 \tag{5.3}$$

as otherwise the descendant rule would simply be subsumed.

5.2.1. Example. Suppose we know that the two rules

$$\begin{array}{rcl} x \neq y,\ y \neq z,\ z \leqslant w & \to & \mathcal{C}, \\ x \neq z,\ z > w & \to & \mathcal{C} \end{array}$$

are correct. Also let us assume that $\textsf{alldifferent}$, a constraint requiring pair-wise difference of its variables, is in the constraint language. By ($\textsf{gen-derive}$) we obtain the new rule

$$\textsf{alldifferent}(x, y, z) \quad \to \quad \mathcal{C},$$

whose condition does not imply, or is implied by, those of the ancestor rules. □

Generally, several possible candidates for the derived condition $\mathcal{B}$ in ($\textsf{gen-derive}$) exist, and they depend on the constraint language. Ideally, a suitable $\mathcal{B}$ can be constructed directly from $\mathcal{A}_1, \mathcal{A}_2$. This is the case for membership rules.

5.3 Transforming Sets of Membership Rules

We specialise the generic meta rules here for the language of membership rules. We refine the meta rules for subsumption and derivation, which allows us to characterise in terms of local consistencies a membership rule set closed under these meta rules.

While according to the definition of a membership rule the body of a rule can consist of multiple inequality constraints, for the purpose of this chapter we can assume that such a rule is decomposed into several membership rules with a single body constraints. So membership rules here are constraint propagation rules in the form

$$C(x_1, \ldots, x_n, y),\ x_1 \in S_1,\ \ldots,\ x_n \in S_n \quad \rightarrow \quad y \neq a,$$

where each S_i is a set of constants, and a is a constant. In the following, C, the constraint associated with the rule, is often irrelevant or clear from the context, and we omit it then from the notation. With the understanding that

$$X = x_1, \ldots, x_n \quad \text{and} \quad S = S_1 \times \cdots \times S_n$$

we write the above membership rule concisely as

$$X \in S \rightarrow y \neq a.$$

We proceed by specialising the subsumption and derivation transformations. Subsequently, we describe the rule set resulting from a stabilising derivation of such transformations. We show that if certain conditions on the source rule set are met then the resulting rule set enforces GAC.

5.3.1 Subsumption

A membership rule can be removed when another one performing the same domain reduction but with wider bounds on the variables is available:

$$\frac{R \cup \{X \in S \rightarrow y \neq a,\ X \in P \rightarrow y \neq a\}}{R \cup \{X \in S \rightarrow y \neq a\}} \qquad \text{if} \quad P \subseteq S \qquad (\mathsf{subsume})$$

It is easy to see that (subsume) is an instance of (gen-subsume): if $P \subseteq S$, then every solution of the constraints $X \in P$ is a solution of the constraints $X \in S$.

5.3.1. Example. $x \in \{2\} \rightarrow y \neq 1$ is subsumed by $x \in \{2, 3\} \rightarrow y \neq 1$. □

5.3.2 Derivation

Two membership rules can sometimes be combined to form another one that allows the same domain reduction in new situations.

$$\frac{R \cup \{X \in S \to y \neq a, X \in P \to y \neq a\}}{R \cup \{X \in S \to y \neq a, X \in P \to y \neq a, X \in Q \to y \neq a\}} \text{ if (5.4)} \quad (\mathsf{derive}_k)$$

$$\begin{array}{ll} (a) & Q_i = S_i \cap P_i \quad \text{for all } i \in [1..n], i \neq k, \\ (b) & Q_k = S_k \cup P_k, \\ (c) & Q_k \supset S_k \text{ and } Q_k \supset P_k, \\ (d) & Q_i \neq \varnothing \quad \text{for all } i \in [1..n]. \end{array} \tag{5.4}$$

These four side conditions guarantee that the derived rule

- inherits correctness from its ancestor rules, (5.4.a) and (5.4.b), where one notices that every solution of $X \in Q$ is a solution of $X \in S$ or $X \in P$ (compare with (gen-derive)),
- is not subsumed by any ancestor rule, (5.4.c),
- is a valid membership rule, (5.4.d).

It is useful to point out how we have used the constraint language underlying the membership rules. The disjunctive constraint "$x_k \in S_k$ or $x_k \in P_k$" can directly be represented in this language, namely as $x_k \in Q_k$.

A (derive_k) step depends on k, and for two ancestor rules there may be several appropriate indices k, satisfying (5.4). Note however, that no derived rule subsumes another with a different k. In the following, when k is not relevant we write just (derive) instead of (derive_k).

5.3.2. Example. From

$$\begin{array}{lll} x_1 \in \{1,2\}, \quad x_2 \in \{1,3\} & \to & y \neq 2, \\ x_1 \in \{2,3\}, \quad x_2 \in \{2,3\} & \to & y \neq 2 \end{array}$$

we obtain

$$\begin{array}{lllll} x_1 \in \{1,2,3\}, & x_2 \in \{3\} & \to & y \neq 2 & \text{with } k = 1, \\ x_1 \in \{2\}, & x_2 \in \{1,2,3\} & \to & y \neq 2 & \text{with } k = 2 \end{array}$$

by (derive_k). □

5.3.3 Result of the Meta Rule Closure

We examine now the properties of exhaustive applications, i. e., closures, of the meta rules (derive), (subsume) for membership rule sets. We proceed in two steps. First, we link the source rule set to the meta rule closure with respect to all correct membership rules associated with the constraint. Subsequently, we characterise the constraint propagation that a closed rule set achieves.

Atomic Rules

5.3.3. Definition. The membership rule $C(X, y), X \in S \to y \neq a$ is *atomic* if each S_i is a singleton set. □

The following note establishes an important 1–1 correspondence between an atomic rule and a non-solution of its associated constraint.

5.3.4. Note. *The atomic rule $C(X, y), X \in S \to y \neq a$ in which the variables X, y have the domains $\mathcal{D}$ is correct if and only if the tuple $d \in \mathcal{D}$ with $\{d[X]\} = S$ and $d[y] = a$ is* not *a solution of C.* □

5.3.5. Example. The tuple $(1, 1, 0)$ is not a solution of the constraint $\mathsf{and}(x, y, z)$ expressing the conjunction $x \wedge y = z$. It corresponds to the correct atomic rules

$$\begin{array}{llll}
\mathsf{and}(x,y,z), & x \in \{1\}, & y \in \{1\} & & \to & z \neq 0, \\
\mathsf{and}(x,y,z), & x \in \{1\}, & & z \in \{0\} & \to & y \neq 1, \\
\mathsf{and}(x,y,z), & & y \in \{1\}, & z \in \{0\} & \to & x \neq 1.
\end{array}$$

□

We denote by *closure*(R) the rule set that results from applying (derive), (subsume) exhaustively. Here is the first observation: all 'interesting' rules are obtained by computing the closure of all atomic rules.

5.3.6. Lemma. *Let R be a set of membership rules, all associated with the constraint C. If R subsumes every* atomic *membership rule correct for C then closure(R) subsumes every membership rule correct for C.*

Proof. We argue by contradiction: Let us say that $r = (X \in S \to y \neq a)$ is correct for C but not subsumed by *closure*$(\mathcal{R})$. Without loss of generality we can assume that r is a most specific such rule, in the sense that all other rules $X \in S' \to y \neq a$ with $S' \subset S$, i. e., subsumed by r, are also subsumed by *closure*$(\mathcal{R})$.

Observe first that r is not atomic. Take then from the condition $X \in S$ some S_k that is not a singleton, and partition it into $S_k = P_k \cup Q_k$ where neither P_k nor Q_k is empty. Construct new rule conditions $X \in P, X \in Q$ by just defining $P_i = Q_i = S_i$ at the remaining indices $i \neq k$.

Since r is a correct rule, so are the rules $X \in P \rightarrow y \neq a$ and $X \in Q \rightarrow y \neq a$. Furthermore, both these rules are strictly subsumed by r, which means they are also subsumed by *closure*(R).

Thus, for each of the two rules, a subsuming rule contained in *closure*$(\mathcal{R})$ exists. Enter these two subsuming rules into (derive). The resulting new rule must subsume r, which contradicts our assumption.

With regard to (subsume), we remark that subsumption is a transitive relation. Therefore, if a rule is subsumed by a rule set then this is still the case after an application of (subsume) to the set. □

We know now which 'seed rules' are necessary so that after closure there are rules for all correct propagations. Next, we establish the local consistency notion achieved by these propagations.

5.3.7. Lemma. *Let R be a set of membership rules correct for their associated constraint C. Let R subsume every rule correct for C. Then the constraint C is closed under R if and only if C is generalised arc-consistent.*

Proof. For the 'if' direction, suppose that the constraint C is closed under R but not under some correct rule $r \notin R$. We show that C is not generalised arc-consistent. Let $r = (X \in S \rightarrow y \neq a)$. We have thus $C[X] \subseteq S$ and $a \in C[y]$.

Since r is a correct rule, we know that for all d in the product of the variable domains we have that $d[X] \in S$ implies $d[y] \neq a$. The counter position is that $d[y] = a$ implies $d[X] \notin S$, and in turn $d[X] \notin C[X]$, for all d.

In other words, the partial instantiation $\{y \mapsto a\}$ can not be extended to a solution of C, so C is not GAC.

For the reverse direction, suppose that $\{y \mapsto a\}$ can not be extended to a solution of the constraint C. So no d exist with $d[y] = a$ and $d[X] \in C[X]$. Then $x_1 \in C[x_1], \ldots, x_n \in C[x_n] \rightarrow y \neq a$ is a correct rule; and as such is subsumed by R. The subsuming rule in R, however, is applicable to C. □

From Atomic Rules to GAC

5.3.8. Definition. Let R be a set of correct membership rules all associated with a constraint C. R is called *atomically complete* with respect to C if R contains or subsumes every correct atomic rule associated with C. □

Here we have the important consequence of Lemmas 5.3.6 and 5.3.7.

5.3.9. Theorem. *Assume that R is atomically complete w. r. t. a constraint C and let $R_{cl} = closure(R)$. R_{cl} is sufficient for enforcing GAC on C; that is, the constraint C is generalised arc-consistent if and only if C is closed under R_{cl}.* □

5.3.4 Infeasible Rules

It is useful to characterise a rule by the following property of its condition.

5.3.10. DEFINITION. A constraint propagation rule is called *feasible* if its condition is satisfiable:

$$\mathcal{A} \rightarrow \mathcal{C} \qquad \text{is feasible if} \qquad Sol(\mathcal{A}, X, \mathcal{D}) \neq \varnothing,$$

where the constraints in $\mathcal{A}$ are on the variables X with domains $\mathcal{D}$. □

For membership rules we find

$$C(X,y), X \in S \;\rightarrow\; y \neq a \qquad \text{is feasible exactly if} \qquad S \cap C[X] \neq \varnothing.$$

Note that an infeasible rule is trivially correct. It is also often redundant.

The notion of membership rules stems from [Apt and Monfroy, 2001] where also the first algorithm for automatically generating such rules is described. We call this generation algorithm RGA here and revisit it in Section 5.6.2.

Since a rule set generated by RGA does not contain infeasible rules but does suffice to establish GAC, one may suspect that infeasible rules are without use. This is not the case, as we see now.

5.3.11. EXAMPLE. The closure-based approach to membership rule generation, unlike RGA, may yield infeasible rules. It may also generate 'partially infeasible' rules. Define the constraint C on the variables x, y with domain $\{1,2,3\}$ as in the following table. The RGA algorithm generates the GAC-enforcing rules $R = \{r_1, \ldots, r_4\}$.

x	y
1	1
3	1
3	3

$$y \in \{3\} \;\rightarrow\; x \neq 1 \qquad (r_1)$$
$$y \in \{1,2,3\} \;\rightarrow\; x \neq 2 \qquad (r_2)$$
$$x \in \{1,2,3\} \;\rightarrow\; y \neq 2 \qquad (r_3)$$
$$x \in \{1\} \;\rightarrow\; y \neq 3 \qquad (r_4)$$

R is a minimal rule set in the sense of the strong redundancy notion in Chapter 4. Consider now the rule

$$y \in \{2\} \rightarrow x \neq 1. \qquad (r_5)$$

It is correct, but infeasible and redundant with respect to R. While R is closed under (derive), (subsume), we can, however, apply (derive) to $R \cup \{r_5\}$, and obtain

$$y \in \{2,3\} \rightarrow x \neq 1, \qquad (r_6)$$

which subsumes both r_1 and r_6. Completing the meta rule closure, we obtain the rule set $R' = R - \{r_1\} \cup \{r_6\}$, which is minimal with respect to redundancy and enforces GAC, just as the original R. However, for constraint propagation, the rule r_6 is preferable to r_1 since its condition is weaker, and R' is therefore preferable to R! □

Including infeasible rules in rule generation by the closure method lets us obtain rules that are more useful for constraint propagation. It is also generally unavoidable for completeness, Theorem 5.3.9, since some atomic rules may be infeasible (as r_5 above).

5.4 Cases of Incremental Rule Generation

We discuss now various useful instances of incremental rule generation, based on the meta rule closure. In each case, we assume that some source constraints $C_1, \ldots, C_m$ with associated rule sets $R_1, \ldots, R_m$ are given. We explain how a new constraint C_{new} is related to the input constraints, i. e.,

$$C_{new} = f_C(C_1, \ldots, C_m),$$

and are interested in obtaining a membership rule set for C_{new} based on the rules for the source constraints,

$$R_{new} = f_R(R_1, \ldots, R_m).$$

We study the requirements for R_{new} to be atomically complete w. r. t. C_{new}, since if that is the case, $closure R_{new}$ is sufficient for enforcing GAC on the constraint C_{new}.

5.4.1 Conjunction of Constraints

Consider two constraints C_1, C_2 on the same variables X to which are associated the rule sets R_1, R_2, resp. We are interested in the conjunctive constraint

$$C_\wedge = C_1 \wedge C_2$$

and a rule set $R_\wedge$ associated with it. In precise notation, we examine the constraint

$$C_\wedge = \langle C_{1R} \cap C_{2R}, X\rangle \quad \text{based on} \quad C_1 = \langle C_{1R}, X\rangle \text{ and } C_2 = \langle C_{2R}, X\rangle.$$

The simple rule set union $R_1 \cup R_2$ does generally not propagate sufficiently to enforce GAC on $C_\wedge$, even if R_i enforces GAC on C_i for both $i = 1, 2$. For example, consider $(x \neq y) \wedge (x = y)$ constraining the variables $x, y \in \{0, 1\}$. It is closed under any rules correct for the one of the constraints $\neq$ and $=$ individually yet it is inconsistent, which GAC-enforcing rules for the conjunctive constraint $(\neq) \wedge (=)$ do detect.

Observe that any atomic rule correct for $C_\wedge$ must also be correct for one or both of C_1, C_2. We define

$$R_\wedge = \mathit{closure}(R_1 \cup R_2),$$

and employ Theorem 5.3.9.

5.4.1. Fact. $R_\wedge$ is atomically complete w. r. t. $C_\wedge$ if R_i is atomically complete w. r. t. C_i for both $i = 1, 2$. □

Relational $(1, m)$-Consistency

The generalisation to conjunctions of m constraints is

$$C_\wedge = \bigwedge_{i=1}^{m} C_i \qquad \text{and} \qquad R_\wedge = \mathit{closure}\left(\bigcup_{i=1}^{m} R_i\right).$$

From the view of the set of the constituent constraints C_i, all on the same set of variables, the local consistency enforced is relational $(1, m)$-consistency [Dechter and van Beek, 1997]. Since we enforce GAC on the conjunctive constraint, an instantiation of any one variable can be extended to a solution of it, which is also a solution of each of the constituent constraints.

Enforcing GAC on the constituent constraints separately is equivalent to relational $(1, 1)$-consistency, a strictly weaker local consistency.

5.4.2. Example. Consider the constraints $\mathsf{and}(x, y, z)$ and $\mathsf{or}(x, y, z)$, representing the logical operators, and their conjunction $c(x, y, z) = \mathsf{and}(x, y, z) \wedge \mathsf{or}(x, y, z)$. It has exactly the two solutions $\{(0, 0, 0), (1, 1, 1)\}$. In the union of the rule sets for $\mathsf{and}, \mathsf{or}$ (such that atomic rules are subsumed as required) we find

$$\begin{array}{rcll} z \in \{1\} & \rightarrow & y \neq 0 & \qquad \text{for } \mathsf{and}, \\ x \in \{1\}, z \in \{0\} & \rightarrow & y \neq 0 & \qquad \text{for } \mathsf{or} \text{ (infeasible rule),} \end{array}$$

which allow to generate the expected rule

$$x \in \{1\} \rightarrow y \neq 0 \qquad \text{for } \mathsf{and} \wedge \mathsf{or}$$

by one step of (derive). □

To generate GAC-enforcing rules for constraints that do not share all variables as required in this section, we need constraint padding.

5.4.2 Constraint Padding

In order to construct the rules for a conjunctive constraint, the participating constraints must be on the same set of variables. This can be achieved by essentially syntactically extending the individual constraints to new variables, without actually constraining them. We call such a modification padding. Extending the rules accordingly is slightly more complicated, due to newly arising infeasible rules.

So we examine $C_p(X, v)$ such that $C_p[X] = C$ and $v \in D_v$. Formally

$$C_p = \langle C_R \times D_v;\ X, v\rangle \quad \text{based on} \quad C = \langle C_R, X\rangle \text{ where } v \text{ not in } X \text{ and } v \in D_v.$$

If R is the set of rules associated with C then the rules R_p associated with C_p are found by

$$R_p = \mathit{closure}(R_1 \cup R_2),$$

where

$$R_1 = \{\, X \in S, v \in D_v \to y \neq a \mid (X \in S \to y \neq a) \in R \,\},$$

$$R_2 = \{\, X \in S, y \in \{a\} \to v \neq b \mid (X \in S \to y \neq a) \in R \ \wedge\ b \in D_v \,\}.$$

The set R_1 pads the input rules by simply adding the redundant test $v \in D_v$. All correct atomic rules with bodies on the variables X of C are constructed in this way. R_1 is closed under $(\mathsf{derive}), (\mathsf{subsume})$ if R is.

The set R_2 consists of rules that disallow values for the new variable v. Since v is not actually constrained, no such rule can exist that is both correct and feasible, however. Therefore, all rules in R_2 are infeasible, since they are correct. Moreover, observe that *every* correct, atomic, infeasible rule with a body disequality on v is subsumed by R_2.

In conclusion, $R_1 \cup R_2$ is atomically complete w. r. t. C_p if R is atomically complete w. r. t. C.

5.4.3. FACT. R_p is atomically complete w. r. t. C_p if R is atomically complete w. r. t. C. □

The pre-closure processing is linear in the size of the set R.

5.4.4. EXAMPLE. We pad the Boolean constraint $\mathsf{not}(x, y)$ with the extra variable $z \in \{0, 1\}$ to $\mathsf{not}'(x, y, z)$.

$$\begin{array}{lll}
\mathsf{not}(x,y), y \in \{0\} \to x \neq 0 & & \mathsf{not}'(x,y,z), y \in \{0\}, z \in D \to x \neq 0 \\
\vdots & & \vdots \\
 & & \mathsf{not}'(x,y,z), x \in \{0\}, y \in \{0\} \to z \neq 0 \\
 & & \mathsf{not}'(x,y,z), x \in \{0\}, y \in \{0\} \to z \neq 1
\end{array}$$

□

Multi-Constraint Membership Rules

We are now in the position to deal with conjunctions of constraints that do not share all variables. To obtain rules for the conjunction from rules of the constraints participating in the conjunction, find the union of all their variables, extend the constraints and their rules by appropriate padding to these variables, and close the union of the resulting rule sets under the meta rules.

This insight enables us to derive multi-constraint membership rules.

5.4.5. EXAMPLE. Let us examine the interaction of the two logic constraints $\mathsf{and}(x, y, z)$ and $\mathsf{not}(x, y)$ in the conjunction $\mathsf{and}(x, y, z) \wedge \mathsf{not}(x, y)$.

Given appropriate rule sets $R_{\mathsf{and}}, R_{\mathsf{not}}$ for the constituent constraints, we proceed by first padding R_{not} by the extra variable z to $R_{\mathsf{not}'}$, as done in Example 5.4.4. Subsequently, $R_{\mathsf{and\_not}}$ is found as $closure(R_{\mathsf{and}} \cup R_{\mathsf{not}'})$. It contains the rule $x \in \{0, 1\}, y \in \{0, 1\} \to z \neq 1$ for the conjunction of $\mathsf{and}, \mathsf{not}$. More precisely, we have derived

$$\mathsf{and}(x, y, z),\ \mathsf{not}(x, y),\ x \in \{0, 1\},\ y \in \{0, 1\}\ \to\ z \neq 1,$$

a *multi-constraint* membership rule. □

In [Abdennadher and Rigotti, 2004], a propagation rule generation method is presented that is capable of producing multi-headed propagation rules directly. The method is based on a generate-and-test approach. Its purpose is the generation of rules for the *interaction* of constraints. For example, one can apply it to $\mathsf{and}(x, y, z)$ and $\mathsf{not}(u, v)$ to generate all rules with these constraints and additional equality constraints between variables from $\{x, y, z\}$ and $\{u, v\}$ in the rule condition.

We can generate equivalent (membership) rules describing all interaction patterns between constraints, by performing the corresponding rule set constructions for each pattern.

Enforcing GAC on conjunctions of constraints instead of just on the participating constraints individually can increase search efficiency, despite the additional propagation cost. That is the case especially when the participating constraints share many variables [Katsirelos and Bacchus, 2001].

5.4.3 Defining a Constraint by its Non-Solutions

While constraints are often defined positively by stating their solutions, sometimes it is more natural to define a constraint *negatively* by stating tuples that are *not* solutions. Suppose that $\langle Neg, X\rangle$ is a set of tuples associated with variables X, and define the constraint $C_N(X)$ by

$$C_N = \langle D^n - Neg, X\rangle \quad \text{where} \quad n = |X| \quad \text{and} \quad X \in D^n.$$

A rule set that is atomically complete w. r. t. C_N can be obtained in a particularly simple way. In fact, we can precisely construct the correct atomic rules, by Note 5.3.4 which states the correspondence between an atomic rule and a non-solution. Abbreviate

$$\mathit{lhs}(X,t,i) = x_1 \in \{t[x_1]\},\ldots,x_{i-1} \in \{t[x_{i-1}]\}, x_{i+1} \in \{t[x_{i+1}]\},\ldots,x_n \in \{t[x_n]\}$$

and define

$$\begin{aligned} R_{\mathit{Neg},i} &= \{\ \mathit{lhs}(X,t,i) \to x_i \neq t[x_i] \mid t \in \mathit{Neg}\ \} \\ R_{\mathit{Neg}} &= \bigcup_{i \in [1..n]} R_{\mathit{Neg},i} \end{aligned} \tag{5.5}$$

The construction of R_{Neg} is linear in the size of Neg: precisely $n \cdot |\mathit{Neg}|$ atomic rules are produced.

We finally have

$$R_N = \mathit{closure}(R_{\mathit{Neg}}).$$

5.4.6. FACT. R_N is atomically complete w. r. t. C_N. □

5.4.7. EXAMPLE. Define a constraint over the variable sequence $x, y, z \in [1..10]$ such that they do *not* represent an increasing sequence x, y, z of prime numbers. The 4 non-solutions are $(2,3,5), (2,3,7), (2,5,7), (3,5,7)$. The respective rules:

12 atomic rules:	8 rules after closure:
$x \in \{2\}, y \in \{3\} \ \to\ z \neq 5$	$x \in \{2,3\}, y \in \{5\} \ \to\ z \neq 7$
$y \in \{3\}, z \in \{5\} \ \to\ x \neq 2$	$x \in \{2\}, y \in \{3,5\} \ \to\ z \neq 7$
$\vdots$	$\vdots$
$y \in \{5\}, z \in \{7\} \ \to\ x \neq 3$	$y \in \{3,5\}, z \in \{7\} \ \to\ x \neq 2$

□

5.4.4 Defining a Constraint by its Solutions

When the constraint is defined positively by an explicit set of solutions (or a procedure that enumerates them), the incremental closure method can be used as well, by first converting the positive definition into a corresponding negative one. The method based on non-solutions, described in the previous section, then becomes applicable.

5.4.8. EXAMPLE. Let C be defined as the Boolean binary constraint $\{(0,1),(1,0)\}$. Its non-solutions are $Neg = \{(0,0),(1,1)\}$. View C to be defined as $\{0,1\}^2 - Neg$ and generate the rules from the negative definition. □

In this way, we have a procedure that corresponds in input and output to the RGA generation algorithm of [Apt and Monfroy, 2001]. We compare the two algorithms in detail in Section 5.6.2.

5.4.5 Enlarging the Base Domain

The following observation explains what it means to extend the variable domains by a new value. We redefine the constraint in such a way that an associated rule set needs not be modified. Assume

$$C = \langle C_R, X\rangle \quad \text{with} \quad X \in D^n.$$

We extend the domain by the value e not previously present. Let $D_e = D \cup \{e\}$, and define

$$C_e = \langle C_R \cup D_e^n - D^n, X\rangle \quad \text{with} \quad X \in D_e^n.$$

So a tuple $t \in D_e^n$ is a solution of C_e either

- if it is already a solution of C, or
- if it uses the new value: $t[x] = e$ for some variable x.

The non-solutions of C_e are exactly the non-solutions of C. Note 5.3.4 entails the following link.

5.4.9. FACT. A given rule set is atomically complete w. r. t. C_e if and only if it is atomically complete w. r. t. C.

Additional Modifications of the New Solution Set

A domain enlargement can be combined well with the addition of solutions or non-solutions that use the new value. The case of adding some non-solutions Neg_e reduces straightforwardly to some known methods. How the non-solutions Neg_e give rise to a constraint is explained in Section 5.4.3. In turn, this constraint is combined with the domain-extended constraint C_e into a conjunctive constraint, following Section 5.4.1. The case of adding some new solutions Pos_e can be reduced to the previous one: the solutions are turned into non-solutions by

$$Neg_e = D_e^n - D^n - Pos_e.$$

Note that the solutions of C are irrelevant.

5.4.10. EXAMPLE. We construct rules for the constraint $\mathsf{or3}$, that represents disjunction in the three-valued logic $0, 1, \mathsf{u}$ [Kleene, 1952]. We base the rule generation on the rules R_{or} for the conventional Boolean constraint or, and we use the fact that we can define $\mathsf{or3}$ alternatively as

$$\mathsf{or3} = \mathsf{or} \cup Pos_{\mathsf{u}}$$
$$Pos_{\mathsf{u}} = \{(0, \mathsf{u}, \mathsf{u}), (\mathsf{u}, 0, \mathsf{u}), (1, \mathsf{u}, 1), (\mathsf{u}, 1, 1), (\mathsf{u}, \mathsf{u}, \mathsf{u})\}.$$

So Pos_{u} defines the value u here. Consider now the constraint or' defined by

$$\mathsf{or}' = \mathsf{or} \cup \{\, (x, y, z) \mid \{\mathsf{u}\} \subseteq \{x, y, z\} \text{ and } x, y, z \in \{0, 1, \mathsf{u}\}\,\},$$

that is, or' permits as a solution *any* tuple containing u at some position. We have extended the or constraint by enlarging the domain by u. If R_{or} is atomically complete w. r. t. or then it is so w. r. t. or' as well; we can write $R_{\mathsf{or}'} = R_{\mathsf{or}}$.

Next, we compute the tuples

$$Neg_{\mathsf{u}} = \{0, 1, \mathsf{u}\}^3 - \{0, 1\}^3 - Pos_{\mathsf{u}}$$

that use the value u and are unacceptable as solutions of $\mathsf{or3}$. We find the corresponding atomic rules in $R_{Neg_{\mathsf{u}}}$ as in (5.5) of Section 5.4.3. Finally, we obtain $R_{\mathsf{or3}} = \mathit{closure}(R_{\mathsf{or}'} \cup R_{Neg_{\mathsf{u}}})$. The rule set R_{or3} is atomically complete w. r. t. the constraint $\mathsf{or3}$. □

5.4.6 Universal Quantification

Assume a constraint C one some variable sequence that includes x. We examine the constraint that results from universally quantifying x.

Let

$$C = \langle C_R;\ X\rangle \quad \text{and} \quad X \in D^n \quad \text{and} \quad X = Y, x.$$

We define a constraint on the same variables except x by

$$C_\forall = \forall x.C = \langle C_{\forall R};\ Y\rangle \quad \text{and} \quad Y \in D^{n-1}$$

such that

$$C_R = \{\, t, a \mid t \in C_{\forall R} \text{ and } a \in D \,\}.$$

So whenever the $(n-1)$-tuple t is a solution of $C_\forall$ then for all values a in the domain of the quantified variable x we have that the n-tuple t, a is a solution to the source constraint C.

Surprisingly, we can obtain rules $R_\forall$ for $C_\forall$ from rules R for C by simply eliminating all references to x:

$$R_\forall = \mathit{closure}(\{\, Z \in S \to y \neq a \mid (Z \in S, x \in S_x \to y \neq a) \in R \,\}).$$

5.4.11. FACT. $R_\forall$ is atomically complete w. r. t. $C_\forall$ if R is atomically complete w. r. t. C. □

PROOF. Consider a rule in R associated with C, and the corresponding non-solution d (Note 5.3.4).

The partial solution $d[Y]$ can clearly not be extended to a full solution by each assignments $\{x \mapsto a\}$ with $a \in D$; the counter example is $a = d[x]$.

So $d[Y]$ is a non-solution of $C_\forall$. This means that we can indeed correctly cut the rules of C down to rules for $C_\forall$.

It remains to consider completeness, that is, whether *all* correct, atomic rules for $C_\forall$ are subsumed. Let $Z \in S \to y \neq a$ be such a rule. But then some rule $Z \in S, x \in S_x \to y \neq a$ must be subsumed by R. For, otherwise all d with $\{d[Z]\} = S, d[y] = a$ were solutions, meaning that x could be all-quantified. □

5.4.12. EXAMPLE. We take the Boolean constraint $\mathsf{or}(x, y, z)$, and quantify universally on x. So we consider

$$\mathsf{or}'(y, z) = \forall x.\, \mathsf{or}(x, y, z).$$

The only solution is $(1, 1)$. Indeed, both $(0, 1, 1)$ and $(1, 1, 1)$ satisfy or.

Two correct rules associated with or are

$$\begin{aligned} &\mathsf{or}(x, y, z), x \in \{0\}, z \in \{1\} \;\to\; y \neq 0, \\ &\mathsf{or}(x, y, z), x \in \{1\}, z \in \{0\} \;\to\; y \neq 0. \end{aligned}$$

They lead to

$$\begin{aligned} &\mathsf{or}'(y, z), z \in \{1\} \;\to\; y \neq 0, \\ &\mathsf{or}'(y, z), z \in \{0\} \;\to\; y \neq 0, \end{aligned}$$

from which by meta rule closure

$$\mathsf{or}'(y, z), z \in \{0, 1\} \to y \neq 0$$

is derived. □

The problem of enforcing GAC on all-quantified constraints is studied in [Bordeaux and Monfroy, 2002]. The authors discuss a number of Boolean constraints and associated rules for enforcing GAC, and point out the need for automatic rule generation for quantified constraints

5.4.7 Existential Quantification

Existential quantification (projection) is the dual to the introduction of variables (padding), Section 5.4.2.

Assume a constraint C one some variable sequence that includes x. We examine the constraint that results from existentially quantifying x.

Presume

$$C = \langle C_R;\ X\rangle \quad \text{and} \quad X \in D^n \quad \text{and} \quad X = Y, x.$$

We define a constraint on the same variables except x by

$$C_\exists = \exists x.C = \langle C_{\exists R};\ Y\rangle \quad \text{and} \quad Y \in D^{n-1}$$

such that

$$C_{\exists R} = \{\, d[Y] \mid d \in C_R \,\}.$$

So whenever d is a solution of $C_\exists$ then a value a in the domain of the quantified variable x exists such that the n-tuple t, a is a solution to the source constraint C.

The construction of rules $R_\exists$ for $C_\exists$ is inverse to the case of all-quantification in the sense that it requires closure *prior* to the modification of the rules:

$$R_\exists = \{\, Z \in S \to y \neq a \mid (Z \in S, x \in D \to y \neq a) \in \mathit{closure}(R) \,\}.$$

5.4.13. FACT. $R_\exists$ is atomically complete w. r. t. $C_\exists$ if R is atomically complete w. r. t. C. Moreover, $R_\exists$ is closed under the meta rules $(\mathsf{derive}), (\mathsf{subsume})$. □

PROOF. Consider a rule of the form $Z \in S, x \in D \to y \neq a$ from the set $\mathit{closure}(R)$. It states that it is correct to conclude $y \neq a$ from $Z \in S$, independent of the value of x. Clearly, the rule $Z \in S \to y \neq a$ is correct for $C_\exists$.

Conversely, consider some correct, atomic rule $Z \in S \to y \neq a$ associated with $C_\exists$. We can conclude from it that *no* solution d of C exist with $\{d[Z]\} = S$ and $d[y] = a$. So the rule $Z \in S, x \in D \to y \neq a$ is correct when associated with C, and consequently it must be subsumed by $\mathit{closure}(R)$.

Finally, notice that $R_\exists$ is closed under $(\mathsf{derive}), (\mathsf{subsume})$, since any transformation possible in $R_\exists$ would have been possible in R, which is closed, using the corresponding ancestor rules. □

5.4.14. EXAMPLE. We take once more the Boolean constraint $\mathsf{or}(x, y, z)$, and quantify now existentially on x, to obtain

$$\mathsf{or}'(y, z) = \exists x.\, \mathsf{or}(x, y, z).$$

The three solutions of or' are $\{(0,0), (0,1), (1,1)\}$. Each can be extended to a solution of or by some x in $\{0, 1\}$.

The only two rules contained in the closure of all atomic rules correct for or and with $x \in D$ in their condition are

$$\begin{aligned}
&\mathsf{or}(x, y, z), x \in D, y \in \{1\} \;\to\; z \neq 0,\\
&\mathsf{or}(x, y, z), x \in D, z \in \{0\} \;\to\; y \neq 1.
\end{aligned}$$

We obtain

$$\begin{aligned}\mathsf{or}'(y,z), y \in \{1\} &\rightarrow z \neq 0,\\ \mathsf{or}'(y,z), z \in \{0\} &\rightarrow y \neq 1.\end{aligned}$$

These two are also the only correct rules for or'. Indeed, the only non-solution $(1,0)$ of $\mathsf{or}'(y,z)$ corresponds to the two rules (Note 5.3.4).

No further (derive) or (subsume) is possible on the rule pair. □

5.5 Example: A Composed fulladder Constraint

In this section we demonstrate how a rule-based GAC-enforcing solver (i. e., a set of membership rules establishing GAC) for a complex constraint can be assembled from the solvers of some base constraints. We use for this the fulladder constraint. It captures the relation linking the binary variables x, y, z, s, c in such a way that the sum of x, y, z is s with the carry bit in c, i. e.,

$$x + y + z \;=\; \overline{c\,s} \;=\; 2c + s \qquad \text{with} \qquad x, y, z, s, c \in \{0, 1\}.$$

The conventional definition using the basic constraints $\mathsf{and}, \mathsf{or}, \mathsf{xor}$ is

$$\begin{aligned}\mathsf{fulladder}(x,y,z,s,c) \quad \equiv \quad \exists\, c_1, c_2, s_1.\;\; & \mathsf{xor}(x,y,s_1) \;\wedge\\ & \mathsf{and}(x,y,c_1) \;\wedge\\ & \mathsf{and}(z,s_1,c_2) \;\wedge\\ & \mathsf{or}(c_1,c_2,c) \;\wedge\\ & \mathsf{xor}(z,s_1,s).\end{aligned}$$

It can be used straightforwardly to construct a rule set for fulladder from rule sets for $\mathsf{and}, \mathsf{or}, \mathsf{xor}$. These, in turn, can be constructed from the corresponding positive or negative definition. We sketch a possible sequence of operations, using the straightforward language in Figure 5.2 (variables are handled informally for simplicity of presentation). The input to this incremental generation are 5 rule sets: one copy of a rule set describing the or constraint on the variables (c_1, c_2, c), and 2 times 2 copies describing $\mathsf{xor}, \mathsf{and}$, on $(x, y, s_1), (z, s_1, s)$ and $(x, y, c_1), (z, s_1, c_2)$, resp.

1. We begin with the first two constraints inside the conjunctive definition of fulladder. Let R_{xor} and R_{and} be the corresponding rule sets for (x, y, s_1) and (x, y, c_1), resp. To construct the rules for the conjunctive constraint

   $$\mathsf{aux\_1}(x,y,s_1,c_1) \;:=\; \mathsf{xor}(x,y,s_1) \wedge \mathsf{and}(x,y,c_1)$$

   it is necessary to pad the constraints to

   $$\mathsf{xor}'(x,y,s_1,c_1) \wedge \mathsf{and}'(x,y,s_1,c_1)$$

and accordingly their rules. We compute

$$
\begin{aligned}
R_{\mathsf{xor}'} &:= \mathsf{pad}(c_1,\ R_{\mathsf{xor}})\\
R_{\mathsf{and}'} &:= \mathsf{pad}(s_1,\ R_{\mathsf{and}})
\end{aligned}
$$

and subsequently

$$R_{\mathsf{aux\_1}} \ := \ \mathsf{union}(R_{\mathsf{xor}'}, R_{\mathsf{and}'})$$

In other words, we obtain $R_{\mathsf{aux\_1}} = R_{\mathsf{xor}'} \cup R_{\mathsf{and}'}$.

2. This pattern of padding and union-building is repeated. Let us denote the result by $R_{\mathsf{aux\_2}}$ associated with the constraint $\mathsf{aux\_2}(x, y, z, s, c, s_1, c_1, c_2)$.

3. It remains to eliminate the auxiliary variables s_1, c_1, c_2 by existential quantification; see Section 5.4.7. The set $R_{\mathsf{aux\_2}}$ must be closed before the appropriate rules can be selected. We obtain

$$R_{\mathsf{fulladder}} \ := \ \mathsf{exists}(\{s_1, c_1, c_2\}, \mathsf{closure}(R_{\mathsf{aux\_2}}))$$

By this process, 94 membership rules for fulladder are constructed from $2 \cdot 12$ rules for both occurrences of xor and $3 \cdot 9$ rules for the two occurrences of and and the single occurrence of or. These input rule are the closure of all correct, atomic rules for their respective constraints, therefore the constructed rule set enforces GAC on fulladder. This in turn means strictly more propagation than GAC on the 5 individual constraints. The CSP $\langle \mathsf{fulladder}(x, y, z, s, c)\,;\ x, z, c \in \{0, 1\}, y \in \{1\}, s \in \{0\}\rangle$ is closed under the subconstraint rules – there is no constraint and thus no rule linking y, s directly. In contrast, one rule constructed for fulladder is $y \in \{1\}, s \in \{0\} \rightarrow c \neq 0$ which allows to propagate to $c \in \{1\}$.

5.6 Implementing the Meta Rule Closure

5.6.1 Uniqueness

To show that the meta rule application strategy has no influence on the result of the closure, we view (derive), (subsume) as a rewrite system. For the relevant background we refer to [Baader and Nipkow, 1998].

5.6.1. LEMMA. *The closure of a membership rule set under* (derive), (subsume) *exists and is unique.*

PROOF. If the number of variables and the size of their base domains is finite, then there are only finitely many syntactically correct rules. Any closure algorithm that applies $(\mathsf{derive}_k), (\mathsf{subsume})$ at most once to any pair of rules and a specific k in the input rule set R terminates.

The meta rule system is confluent. We prove this by showing that every critical pair is joinable. The rule sets R_1, R_2 in a critical pair $\langle R_1, R_2 \rangle$ are the respective results of applying two meta rules to the same source rule set.

Joinability of critical pairs is easy to verify for pairs stemming from (subsume)+(subsume) and (derive)+(derive). For the meta rules (subsume)+(derive), the interesting case is the critical pair arising from the source rule set $R \cup \{r_a, r_b, r_c\}$ and where r_a subsumes r_b, and r_b, r_c have a descendant rule r_d. An application of the subsumption meta rule can here prevent a subsequent application of the derivation meta rule. We find the two possible initial derivations

$$\frac{R \cup \{r_a, r_b, r_c\}}{R \cup \{r_a, r_c\}}\ (\mathsf{subsume})\text{ on } r_a, r_b \qquad \frac{R \cup \{r_a, r_b, r_c\}}{R \cup \{r_a, r_b, r_c, r_d\}}\ (\mathsf{derive})\text{ on } r_b, r_c$$

so the critical pair is $\langle R \cup \{r_a, r_c\},\ R \cup \{r_a, r_b, r_c, r_d\}\rangle$. We show this pair to be joinable by a case distinction on whether r_a subsumes r_d.

r_a **subsumes** r_d**.** The two derivations (continuing the ones above)

$$R \cup \{r_a, r_c\} \qquad\qquad \cfrac{\cfrac{R \cup \{r_a, r_b, r_c, r_d\}}{R \cup \{r_a, r_c, r_d\}}\ (\mathsf{subsume})\text{ on } r_a, r_b}{R \cup \{r_a, r_c\}}\ (\mathsf{subsume})\text{ on } r_a, r_d$$

show that $R \cup \{r_a, r_c\}$ and $R \cup \{r_a, r_b, r_c, r_d\}$ are joinable.

r_a **does not subsume** r_d**.** In this case, from r_a and r_c a descendant r_e can be derived. In turn, r_e subsumes r_d. For the critical pair, we have the following two continuations of the above derivations:

$$\frac{R \cup \{r_a, r_c\}}{R \cup \{r_a, r_c, r_e\}}(\mathsf{derive})\text{ on } r_a, r_c \qquad \cfrac{\cfrac{\cfrac{R \cup \{r_a, r_b, r_c, r_d\}}{R \cup \{r_a, r_c, r_d\}}\ (\mathsf{subsume})\text{ on } r_a, r_b}{R \cup \{r_a, r_c, r_d, r_e\}}\ (\mathsf{derive})\text{ on } r_a, r_c}{R \cup \{r_a, r_c, r_e\}}\ (\mathsf{subsume})\text{ on } r_e, r_d$$

So $R \cup \{r_a, r_c\}$ and $R \cup \{r_a, r_b, r_c, r_d\}$ are joinable in $R \cup \{r_a, r_c, r_e\}$.

Since the rewrite system $(\mathsf{derive}), (\mathsf{subsume})$ is terminating and confluent, the closure of a rule set under these meta rules exists and is unique. □

Lemma 5.6.1 allows us to apply the (subsume) meta rule *eagerly* when computing the closure, which improves convergence.

$\boxed{\mathsf{RGA}}$: constraint $C \subseteq D^n$ on $X \longmapsto$ rule set enforcing GAC on C

$R := \varnothing$
for each $V \subset X$ in increasing order **do**
  for each $S \subseteq D^{|V|}$ in decreasing order,
    where $S_i \subseteq C[v_i]$ and $S \cap C[V] \neq \varnothing$, **do**
    for each $y \in X - V$ and each $d \in D$ **do**
      let r be the rule $C(X), V \in S \rightarrow y \neq d$
      if r is correct and not subsumed by R **then**
        $R := R \cup \{r\}$
      end
    end
  end
end
return R

Figure 5.1: Original rule generation algorithm RGA [Apt and Monfroy, 2001]

5.6.2 Relation to the Original Generation Algorithm for Membership Rules

We have already seen in Section 5.3.4 that RGA sometimes does not find every 'interesting' rule. We inspect here in detail its relation to the closure-based approach to rule generation, which can be used in the same way as RGA to generate rules, by the method described in Section 5.4.4.

RGA implements essentially a generate-and-test approach, where the rule candidates are ordered by subsumption such that the output rule set grows steadily as unsubsumed rules are added. We quote RGA in Figure 5.1.

5.6.2. LEMMA. *For a given constraint C, denote by R_{RGA} the rule set that* RGA *generates. Let R_{cl} be a set of correct rules associated with C such that every atomic rule correct for C is subsumed, and R_{cl} is closed under* (derive), (subsume)*. Then*

- *every rule in R_{RGA} is subsumed by some rule in R_{cl}, and*
- *every feasible rule in R_{cl} subsumes some rule in R_{RGA}.*

PROOF. RGA enumerates all correct, feasible rules, discarding those that are subsumed. In turn, R_{cl} subsumes all correct rules, by Theorem 5.3.9.

Inversely, each feasible rule in R_{cl} is correct, and not subsumed by a different correct rule. This means that it is either contained in R_{RGA}, or subsumes a rule therein. The latter case arises due to infeasible rules and (derive). □

rule-set	::=	positive_definition(⟨tuple-set⟩)
	|	negative_definition(⟨tuple-set⟩)
	|	pad(⟨new-variable⟩, ⟨rule-set⟩)
	|	enlarge_domain(⟨new-value⟩, ⟨rule-set⟩)
	|	exists(⟨variable⟩, ⟨rule-set⟩)
	|	for_all(⟨variable⟩, ⟨rule-set⟩)
	|	union(⟨rule-set⟩, ⟨rule-set⟩)
	|	closure(⟨rule-set⟩)

Figure 5.2: A language for incremental membership rule generation

5.7 Implementation and Empirical Evaluation

We implemented a prototype of incremental rule generation in the ECL^{i}PS^e system. The program accepts rule generation requests in the language described in Fig. 5.2 (where, for simplicity, domain and variable handling is omitted). The rule set closure is computed by the algorithm shown in Fig. 5.3. We argue for its correctness briefly and informally by stating that R in the algorithm remains always closed under (derive), (subsume), and that any possible (derive) between two rules of R is collected (stage-wise) in $\mathcal{D}$. Since upon termination $\mathcal{D}$ has been emptied into R by addition or subsumption, we then have that R is closed under (derive), (subsume) and that all rules of the input rule set are subsumed by R.

Benchmarks

We examined the behaviour of the closure algorithm for the generation of rule sets from positive, extensional constraint definitions. This enables a direct comparison with the RGA algorithm of [Apt and Monfroy, 2001].

We used random constraints with uniformly distributed solutions. We varied the tightness of the constraint, i. e. the proportion of non-solutions (a small tightness means many solutions), and we examined varying arity and domain size. Our random constraint generator is based on the program [Bessière, 1996] which was adapted so as to generate a single, n-ary constraint definition. Per data point we used 5 random constraints and 3 repetitions for each (we found only small variances in the measured times).

The results, summarily reported in Fig. 5.4, indicate that the closure-based rule generation approach is more efficient than RGA by orders of magnitudes when the constraint arity or the tightness is small.

$\boxed{\textsf{closure}}$: rule set $R \longmapsto closure(R)$

return $\bigcup_{\forall y,\forall a}$ closure_split($\{ r \in R \mid r = (h \to y \neq a)$ for some $h \}$)

$\boxed{\textsf{closure\_split}}$: rule set $R \longmapsto closure(R)$

assumes that all rules have the same body

if $R = \varnothing$ **then return** $\varnothing$
else
  choose $r \in R$
  return closure_add($R - \{r\}, \{r\}$)
end

$\boxed{\textsf{closure\_add}}$: rule sets $\mathcal{A}, R \longmapsto closure(\mathcal{A} \cup R)$

assumes that R is closed

if $\mathcal{A} = \varnothing$ **then return** R
else
  $\mathcal{D} := \varnothing$
  for each $r \in \mathcal{A}$ not subsumed by R **do**
    $\langle \mathcal{D}_r, R \rangle :=$ closure_add_one(r, R)
    $\mathcal{D} := \mathcal{D} \cup \mathcal{D}_r$
  end
  return closure_add($\mathcal{D}, R$)
end

$\boxed{\textsf{closure\_add\_one}}$: rule r, rule set $R \longmapsto$ rule set pair $\langle \mathcal{D}, R' \rangle$

$\mathcal{D}$ is the set of descendants between r and R, and
R' is R updated with r; always considering subsumption

delete from R all rules subsumed by r
$\mathcal{D} :=$ all (derive) descendants of r and any rule in R
if some $r' \in \mathcal{D}$ subsumes r **then**
  return closure_add_one(r', R)
else
  delete from R all rules subsumed by $\mathcal{D}$
  return $\langle \mathcal{D}, R \cup \{r\} \rangle$
end

Figure 5.3: Algorithm to close a membership rule set under (derive), (subsume)

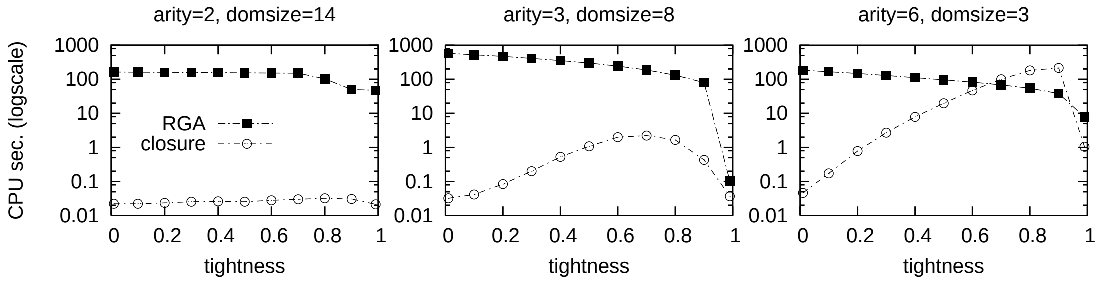

Figure 5.4: Rule generation from positive random constraint definitions

5.8 Final Remarks

We presented an incremental approach to the automatic generation of constraint propagation rules; and we examined it in depth for the case of membership rules. The closure of a rule set under two meta rules constitutes the core of this method. We showed that various ways of defining a constraint incrementally have a corresponding rule generation method. We could also demonstrate the efficiency of this method. That, and the description by meta rules, may make it possible to include it as a source-to-source transformation process during the compilation of `CHR` rules [Holzbaur, 2002].

An important contribution of this work is that it helps to further explain rule-based constraint propagation. For the class of membership rules, the close connection between an atomic unit of the constraint definition, i. e. a single non-solution, and a GAC-enforcing rule application is visible. Evidence for the importance of the explanatory aspect of the rule-based view on constraint propagation is the recent work [Choi et al., 2003], in which propagation rules are employed as a means to argue for operational relevance or redundancy of constraints in dual-model CSPs.

While a complete meta rule closure of a rule set subsuming all correct, atomic rules is necessary to obtain a GAC-enforcing propagation operator for a constraint, compromising on these conditions (except for correctness) does not impede correctness of the corresponding propagation. Situations are conceivable in which it is useful to enter only some of the atomic rules into the closure, or in which the closure is executed incompletely. The resulting solver generally propagates to a local consistency weaker than GAC, but it may consist of less, simpler rules and may therefore be faster to execute – a common trade-off in constraint programming.

Finally we note that dynamic updating of solvers may be of interest in "open-world" constraint satisfaction [Faltings and Macho-Gonzalez, 2002], where gathering the constituents of a CSP, e. g. the tuples defining a constraint, is part of the problem solving process. This is the case, for example, when a CSP is represented in a distributed way on several internet sites.

Chapter 6

Constraint-Based Automatic Test Pattern Generation

6.1 Introduction

As a demonstration of using membership rules for constraint propagation, we consider the problem of automatic test pattern generation for sequential circuits.

The production process of modern electronic integrated circuits is very complex. As a consequence, in practice defective circuits are produced. Subsequent testing is used to filter them out. Since it is impossible to test circuits completely, only some types of faults are targeted. The most common fault model for digital circuits is called *stuck-at* fault. A circuit given in its decomposition into primitive logical gates has a single stuck-at fault if it behaves as if one input or output line of gate was cut and replaced by a permanent value 0 or 1. This fault model covers many other faults occurring in digital circuits.

Testing for stuck-at faults consists in setting the circuit input to defined values and checking if the observed output coincides with the output expected from the circuit specification. ***Automatic test pattern generation*** (ATPG) is the problem of constructing an exhaustive set of test input patterns for the possible stuck-at faults in a circuit, given its specification. A problem introduction and extensive survey of solution algorithms is provided by [Cheng, 1996]. A general introduction to digital electronic circuits is [Shiva, 1988], for example.

For combinational circuits, which have no internal state, constraint-based approaches to ATPG [Simonis, 1989, Hentenryck et al., 1992] and related problems [Azevedo, 2003] exist. Sequential (stateful) circuits have a strictly more complex structure, however, and while some ideas carry over, the mentioned combinational approaches as such are not applicable. One point of increased difficulty is the description of the constraint propagation. We show how rule-based methods, in particular the automatic generation of membership rules, can be used.

We introduce ATPG first in detail for combinational circuits, and move then to sequential circuits, before discussing constraint-based methods.

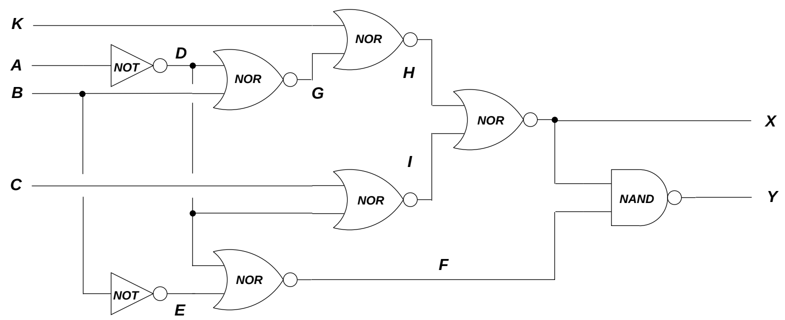

Figure 6.1: Combinational circuit C in gate-level detail

6.1.1 Combinational Circuits

A combinational circuit can be viewed as a function that maps tuples of Booleans representing the input signals of the circuit to tuples of Booleans representing the output signals:

$$O = \mathsf{circuit}(I) \qquad \text{where} \qquad I = \langle I_1, \ldots, I_{n_{\text{in}}} \rangle \quad \text{and} \quad O = \langle O_1, \ldots, O_{n_{\text{out}}} \rangle,$$

and $I_i, O_j \in \{0, 1\}$ for all indices i, j. The function $\mathsf{circuit}$ is composed of the basic Boolean connectives and, or, not, nand, nor, with auxiliary variables to represent internal circuit signal lines. This decomposition corresponds to the gate-level specification of the circuit. For simplicity, we assume gates with two inputs from now on.

6.1.1. Example. Consider the circuit C in Fig. 6.1, adapted from [Muth, 1976]. It corresponds to the function

$$\langle X, Y \rangle = \mathsf{circuit}_{\mathsf{C}}(A, B, C, K).$$

As its decomposition we have the set of atomic operations

$$\begin{array}{lll} E = \mathsf{not}(B), & G = \mathsf{nor}(D, B), & X = \mathsf{nor}(H, I), \\ F = \mathsf{nor}(D, E), & H = \mathsf{nor}(K, G), & Y = \mathsf{nand}(K, F), \\ D = \mathsf{not}(A), & I = \mathsf{nor}(D, C). & \end{array}$$

□

Faults and Tests

A ***stuck-at-*`X`** ***fault*** in a circuit is a signal line that does not pass through its incoming value but instead the constant value `X`, which is either 0 or 1. The short-cuts `sa0` and `sa1` are often used. In terms of the decomposition, the fault

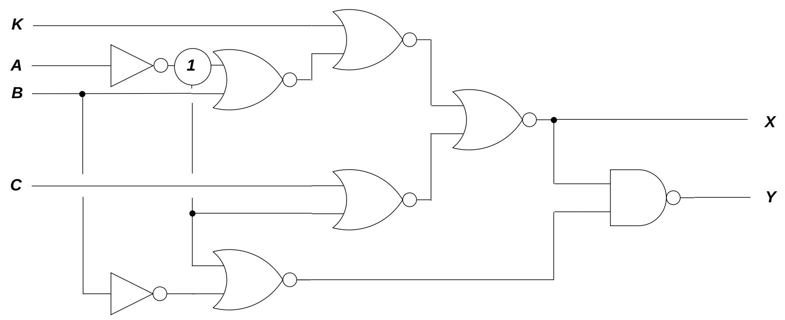

Figure 6.2: Stuck-at-1 fault at line D in circuit C

$\mathtt{saX}$ occurring at the input or output line of gate can be explained by the following modification.

$$\begin{array}{lll} \text{Replace} & O = \mathsf{gate}(I_1, \ldots, I_k) & \\ \text{by} & O = \mathsf{gate}(I_1, \ldots, I_{i-1}, \mathtt{X}, I_{i+1}, \ldots, I_k) & \text{if input } I_k \text{ faulty,} \\ \text{or by} & O = \mathtt{X}, & \text{if output faulty.} \end{array} \qquad (6.1)$$

Given a circuit and its version with a specific stuck-at fault, represented respectively by the functions

$$O_G = \mathsf{circuit}(I) \qquad \text{and} \qquad O_F = \mathsf{faulty\_circuit}(I),$$

a ***test pattern*** is a Boolean vector $tp = \langle tp_1, \ldots, tp_{n_{\text{in}}} \rangle$ such that

$$\mathsf{circuit}(tp) \neq \mathsf{faulty\_circuit}(tp).$$

So the test pattern makes the fault observable at the output. Locating the fault is not required.

Two stuck-at faults can correspond to each other; for example, an $\mathtt{sa0}$ fault at an input line of an and gate cannot be distinguished by any test input from $\mathtt{sa0}$ at the gate output. These correspondences can be pre-computed, and testing needs only take place for the *collapsed fault set*. A stuck-at fault can also be such that no test input at all exists that makes it observable. Such faults are often called *redundant* in the literature; this is to distinguish them from *undetectable* faults, for which the concrete test generation method at hand finds no test (i. e., it is incomplete).

While a single test cannot uncover all stuck-at faults in a circuit, it often can test for more than one fault. Hence, the objective of test pattern generation is to generate a small set of test patterns covering as many faults as possible.

6.1.2. Example. Fig. 6.2 shows circuit C with a stuck-at-1 fault introduced at line D. The circuit decomposition corresponds here to

$$\begin{aligned} E &= \mathsf{not}(B), & G &= \mathsf{nor}(D, B), & X &= \mathsf{nor}(H, I), \\ F &= \mathsf{nor}(D, E), & H &= \mathsf{nor}(K, G), & Y &= \mathsf{nand}(K, F), \\ D_{\text{faulty}} &= \mathsf{not}(A), & I &= \mathsf{nor}(D, C), \\ D &= 1. \end{aligned}$$

The tuple $\langle 1, 1, 1, 1 \rangle$ is a test pattern for this fault. We find

$$\begin{aligned} \mathsf{circuit}_{\mathsf{C}}(1, 1, 1, 1) &= \langle 1, \mathbf{0} \rangle, \quad \text{but} \\ \mathsf{faulty\_circuit}_{\mathsf{C}}(1, 1, 1, 1) &= \langle 1, \mathbf{1} \rangle. \end{aligned}$$

□

Complexity. Test pattern generation for combinational circuits is computationally costly; it is an NP-complete problem. It is not difficult to see that the SAT problem can be reduced to it [Papadimitriou, 1994]. A SAT instance, i. e. a set of propositional variables and a formula in CNF with clauses (disjunctions) over them, can be viewed as a circuit whose inputs correspond to the variables, and whose single output reflects the truth of the formula. Satisfiability corresponds to the existence of a test pattern for a stuck-at-0 fault at the circuit output. To see membership in NP, assume that a test pattern for a specific fault is given. The output of the correct circuit with the test pattern as its input can be computed in time polynomial in the size of the circuit decomposition. The same holds for the faulty circuit, which differs only in one place from the correct circuit. A comparison of both output vectors takes time linear in their size.

6.1.2 Sequential Circuits

Sequential circuits are based on combinational ones but additionally have *inaccessible* internal memory elements and feedback loops. The output of such a circuit is a function of its input and its current state, and the next state is a function of the input and the current state. We consider here synchronous sequential circuits, whose behaviour depends on the signal values at discrete time points, controlled by a clock tick. Roughly, the output signals are valid *at* the clock ticks, and the computation and the memory update take place *between* clock ticks.

So a sequential circuit computes a time-dependent function $O = \mathsf{seq\_circuit}(I)$ that is *recursively* defined. We can describe it by a combinational circuit

$$\langle O, M' \rangle = \mathsf{circuit}(I, M)$$

where $M = \langle M_1, \ldots, M_{n_{\text{mem}}} \rangle$ is the state, that is, the contents of the memory elements, at the current clock tick. M' is the state at the following clock tick.

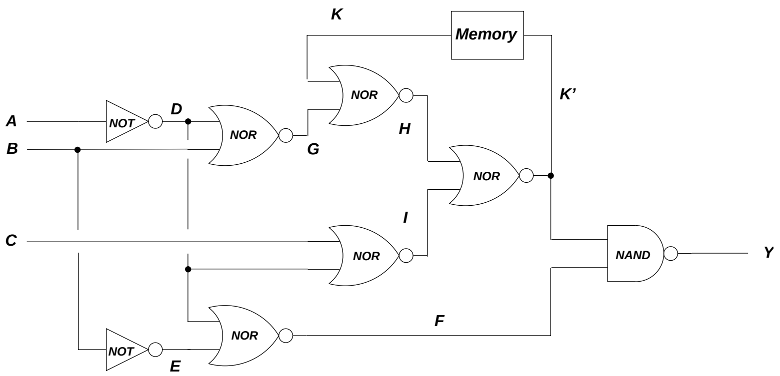

Figure 6.3: Sequential circuit S

M, M' are referred to as the ***pseudo-inputs*** and ***pseudo-outputs***, resp., indicating that they cannot be controlled or observed directly.

6.1.3. EXAMPLE. Consider circuit S in Fig. 6.3, taken from [Muth, 1976]. Its underlying combinational circuit is C from the previous examples. The sequential circuit S corresponds to the functions

$$\langle Y\rangle = \mathsf{seq\_circuit}_{\mathsf{S}}(A, B, C), \qquad \text{and} \qquad \langle Y, K'\rangle = \mathsf{circuit}_{\mathsf{S}}(A, B, C, K),$$

respectively. The decomposition of S is

$$\begin{aligned} E &= \mathsf{not}(B), & G &= \mathsf{nor}(D, B), & K' &= \mathsf{nor}(H, I),\\ F &= \mathsf{nor}(D, E), & H &= \mathsf{nor}(K, G), & Y &= \mathsf{nand}(K', F),\\ D &= \mathsf{not}(A), & I &= \mathsf{nor}(D, C). \end{aligned}$$

□

Faults and Tests

A stuck-at fault in a sequential circuit corresponds to a stuck-fault in its underlying combinational circuit. Sequential circuit testing, however, differs from combinational testing in that only the proper output signals can be observed, not the internal state. Moreover, one is interested also in faults that need several clock cycles to manifest themselves at an output line. Consequently, a test for a sequential circuit consists of a *sequence of input vectors*, which propagate the fault effect to an output line step by step *independently of the initial state.*

Let us make this formal. We make the internal state M explicit in $\mathsf{seq\_circuit}$, and we extend it to a function on input sequences:

$$\begin{aligned} \mathsf{seq\_circuit}(\langle I\rangle, M) &= O \qquad \text{where} \quad \mathsf{circuit}(I, M) = \langle O, M'\rangle,\\ \mathsf{seq\_circuit}(\langle I_1, I_2, \ldots, I_k\rangle, M) &= \mathsf{seq\_circuit}(I_2, \ldots, I_k, M')\\ &\qquad\qquad \text{where} \quad \mathsf{circuit}(I_1, M) = \langle O_1, M'\rangle. \end{aligned}$$

input				output		time
A	B	C	K	Y	K'	
1	0	1	?	1	1/?	1
1	1	1	1/?	**0/1**	1/?	2

Table 6.1: Testing for `sa1` at D in circuit S

$\mathsf{circuit}$ denotes the combinational portion of $\mathsf{seq\_circuit}$.

A test pattern (for a specific fault) is now a finite sequence T of input vectors that makes the fault observable independently of the initial internal state:

$$\forall m \in \{0,1\}^{n_{\mathrm{mem}}}. \ \ \mathsf{seq\_circuit}(T,m) \ \neq \ \mathsf{faulty\_seq\_circuit}(T,m).$$

(Recall that n_{mem} is the number of internal memory elements.) $\mathsf{faulty\_seq\_circuit}$ is a variant of $\mathsf{seq\_circuit}$ with a stuck-at fault in its combinational portion. Note that we are faced with *repeated* fault effects since the fault is present at *every* clock tick.

6.1.4. Example. Take circuit S in Fig. 6.3 of Example 6.1.3, and assume a stuck-at-1 fault at line D (so the underlying combinational circuit is as in Fig. 6.2). The test pattern $\langle\langle 1,0,1\rangle,\langle 1,0,1\rangle\rangle$ consisting of two input vectors makes `sa1` observable at D.

The behaviour of the circuit is indicated in Table 6.1. We denote uncertain values, such as the initial state, by '?'. When the signal values in correct (G) and faulty (F) circuit deviate, we use the notation G/F.

After the input $\langle 1,1,1\rangle$ in the second step, the output Y is 0 for the correct circuit but 1 for the faulty one, making the fault observable. It is remarkable that initialising the memory element K to a known value is not needed in each case; it suffices to do so in the correct circuit. □

6.2 Modelling ATPG with Constraints

Test pattern generation can be formulated in terms of constraint satisfaction problems. For combinational circuits this work has been done; we begin by reviewing these approaches. After that we present several models for sequential circuits.

We assume that the input to a test generation procedure is a circuit defined by its decomposition, and the location and type of a single stuck-at fault. The desired output is a test pattern for this fault.

or_4	0	1	d	$\overline{\mathsf{d}}$
0	0	1	d	$\overline{\mathsf{d}}$
1	1	1	1	1
d	d	1	d	1
$\overline{\mathsf{d}}$	$\overline{\mathsf{d}}$	1	1	$\overline{\mathsf{d}}$

Table 6.2: Disjunction in the 4-valued logic

6.2.1 Combinational ATPG

The 4-valued Model

Correct and faulty circuit are structurally almost identical. This important observation is used in the model underlying the D-Algorithm of [Roth, 1966], where both circuit models are overlapped. Every gate common to both circuits and defined on the Booleans is lifted to the Cartesian product of the Booleans. In the notation G/F where G is the value of the correct ('good') circuit and F the value of the faulty circuit, a gate in this 4-valued model is thus defined by

$$\mathsf{gate}_4(X_G/X_F,\ Y_G/Y_F,\ Z_G/Z_F) \ = \ \mathsf{gate}(X_G, Y_G, Z_G) \text{ and } \mathsf{gate}(X_F, Y_F, Z_F).$$

The notation of the combined values is conventionally simplified by

$$0/0 = 0, \qquad 1/0 = \mathsf{d},$$
$$1/1 = 1, \qquad 0/1 = \overline{\mathsf{d}}.$$

So we deal with operations over the domain $D_4 = \{0, 1, \mathsf{d}, \overline{\mathsf{d}}\}$. The definition of the or-gate in this 4-valued logic, for example, is given in Table 6.2. We use combined_circuit to describe the function of the correct and faulty circuit considered together in the 4-valued logic. A test pattern tp is now such that

$$\mathsf{combined\_circuit}(tp) = \langle o_1, \ldots, o_{n_{\mathrm{out}}} \rangle \quad \text{with} \quad o_i \in D_4$$

and

$$o_k = \mathsf{d} \qquad \text{or} \qquad o_k = \overline{\mathsf{d}}$$

for some k.

Modelling as a CSP. Test pattern generation can be expressed as a constraint satisfaction problem with the 4-valued logic. The first approach is reported in [Simonis, 1989], but it is too restrictive; see the critique in [Azevedo, 2003, p. 32].

Here is an alternative modelling of ATPG as a CSP.

Variables. Every input, output and internal signal of the circuit corresponds to a single variable.

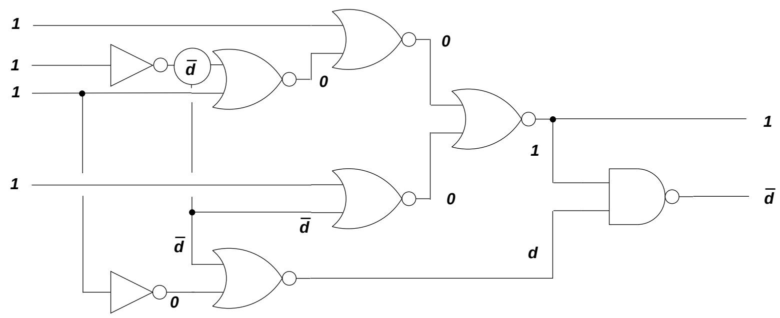

Figure 6.4: Testing for stuck-at-1 with the 4-valued logic

Domains. Input variables have the domain $\{0, 1\}$, all other variables range over $D_4 = \{0, 1, \mathsf{d}, \overline{\mathsf{d}}\}$.

Constraints.

- Every circuit gate is represented by the corresponding constraint in the 4-valued logic.
- The gate with the fault at its input or output is modified as explained in (6.1) of Section 6.1.1, but with X is replaced by d if the fault is stuck-at-0, and by $\overline{\mathsf{d}}$ if the fault is stuck-at-1.
- The output variables are constrained to show the fault: at least one variable must take the value d or $\overline{\mathsf{d}}$.

6.2.1. Example. Fig. 6.4 shows circuit C with a stuck-at-1 fault at line D. The values at the signal lines show the situation that is obtained by the test pattern $\langle 1, 1, 1, 1\rangle$, using the 4-valued logic. □

6-valued Model

Considering the distribution of the values in the circuit for a test pattern, one can observe that the fault effect traces a path from the fault site to some circuit output. In the circuit C of Figures 6.1 and 6.4, the fault effect travels from $D = \overline{\mathsf{d}}$ via $F = \mathsf{d}$ to $Y = \overline{\mathsf{d}}$, for instance. There can be several such paths as the fault effect can multiply via a signal line entering several gates. However, one path must exist. Conversely, a fault effect can also 'disappear' at a gate; take for instance $\mathsf{or}(\mathsf{d}, 1) = 1$ that prevents d from being observable at the output.

These observations, the necessary existence of a fault effect path and the possibility of disappearing fault effects, led to a refined model. The idea of [Hentenryck et al., 1992] is to distinguish two pairs of fault effect values,

- $\mathsf{d}, \overline{\mathsf{d}}$ for the single necessary fault effect path, and
- $\mathsf{e}, \overline{\mathsf{e}}$ for 'expendable' fault effects, which may disappear.

or_6	0	1	e	$\bar{\mathsf{e}}$	d	$\bar{\mathsf{d}}$
0	0	1	e	$\bar{\mathsf{e}}$	d	$\bar{\mathsf{d}}$
1	1	1	1	1		
e	e	1	e	1	d	
$\bar{\mathsf{e}}$	$\bar{\mathsf{e}}$	1	1	$\bar{\mathsf{e}}$		$\bar{\mathsf{d}}$
d	d		d			
$\bar{\mathsf{d}}$	$\bar{\mathsf{d}}$			$\bar{\mathsf{d}}$		

fanout		
0	0	0
1	1	1
e	e	e
$\bar{\mathsf{e}}$	$\bar{\mathsf{e}}$	$\bar{\mathsf{e}}$
d	d	e
d	e	d
$\bar{\mathsf{d}}$	$\bar{\mathsf{d}}$	$\bar{\mathsf{e}}$
$\bar{\mathsf{d}}$	$\bar{\mathsf{e}}$	$\bar{\mathsf{d}}$

Table 6.3: Disjunction and fan-out in the 6-valued logic

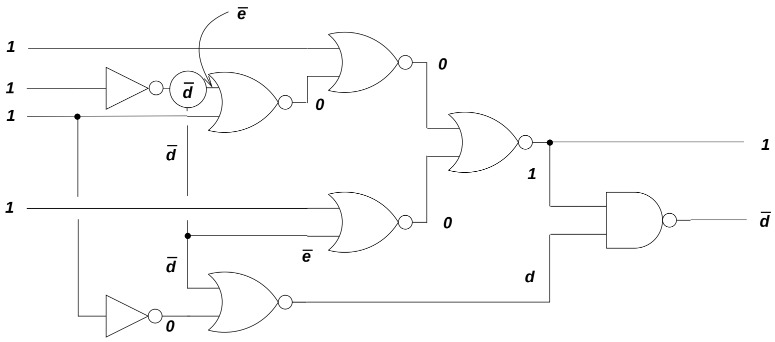

Figure 6.5: Testing for stuck-at-1 with the 6-valued logic

The resulting logic is 6-valued. Table 6.3 shows the or constraint, viewed as a function. It is not a total function anymore: not all input combinations with d-values (d or $\bar{\mathsf{d}}$) are permitted. A d-value is forced to the output, and at most one d-value is expected at the input. The result is a single fault effect path from input to output.

A new concern is the distribution of the fault effect through fan-out points. For example, in the example circuit, the line D enters three nor gates. A special fan-out constraint is introduced for this purpose. Consider a line used as an input to two gates. This situation is modelled by the line variable O and the two input variables I_1, I_2, constrained by

$$\mathsf{fanout}(O, I_1, I_2).$$

The definition is given in Table 6.3. It forces the non-expendable fault effect $\mathsf{d}, \bar{\mathsf{d}}$ to take exactly one path, passing it on as an expendable fault effect on the other. All other values are simply distributed. The fan-out constraint and the 6-valued logic constraints force existence of a single fault effect path from input to output.

6.2.2. Example. Fig. 6.5 shows test generation with the 6-valued logic (the

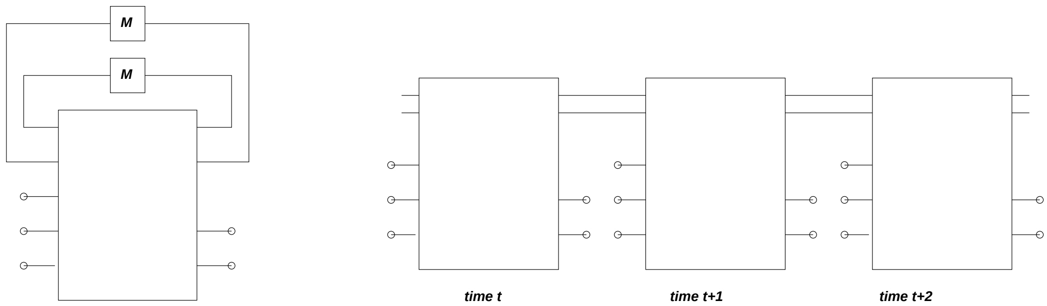

Figure 6.6: Unfolding a sequential circuit into time-frames

two fan-out points are left implicit). Note the single path of d-values, and the disappearing e-values. □

6.2.2 Sequential ATPG

Sequential circuits add two new issues to test pattern generation. Recall that a test pattern is a sequence $T = \langle t_1, \ldots, t_{n_{\mathrm{tp}}} \rangle$ of input vectors t_i such that

$$\forall m \in \{0,1\}^{n_{\mathrm{mem}}}.\ \ \mathsf{seq\_circuit}(T, m)\ \neq\ \mathsf{faulty\_seq\_circuit}(T, m) \qquad (6.2)$$

where n_{mem} is the number of internal memory elements.

So a sequence of a priori unknown length must be found, and the problem statement contains universal quantifications. We deal with these issues separately.

Sequences of Input Vectors

A test sequence of shortest length can be found by iteratively searching over sequences of specific length, starting with length one and increasing it in successive rounds.

The subproblem of finding a test sequence of specific length can be dealt with by ***unfolding*** the sequential circuit into the sequence of its combinational portions, usually called ***time-frames***; see Fig. 6.6. The memory elements are replaced by new lines linking adjacent time-frames. These new lines connect the pseudo-inputs and pseudo-outputs left after removing the memory elements. Unfolding is a general technique of the algorithms surveyed in [Cheng, 1996].

A test pattern assigns the inputs of each time-frame. Some output of the last time-frame shows the fault effect. The fault effect traces a path to the output in this setting as well. Since the fault occurs in *each* time-frame, there is a specific time-frame at whose fault site the fault effect path starts. This time-frame is not necessarily the first one; some initial test input vectors may be needed to *activate* the fault, e. g., to propagate a signal 1 to a stuck-at-0 fault site. The observation that such a fault-activating time frame exists, can be used to guide the iterative search over test patterns of fixed length. For example, the fault can be activated

or_3	0	1	u
0	0	1	u
1	1	1	1
u	u	1	u

Table 6.4: Disjunction in the 3-valued logic

in the initial time-frame and new time-frames can be added forward or backward in time, up to the current maximum number of time-frames.

An obvious desirable property of a sequence of test input vectors is that ***state repetition*** is avoided, that is, the induced individual states should be pair-wise different. A state repetition in a test implies that an equivalent but shorter test exists; the intermediate input sequence is redundant and can be extracted.

By unfolding a sequential circuit into time-frames we can thus transform Problem (6.2) into sequences of problems of the form

$$\forall m \in \{0,1\}^{n_{\mathrm{mem}}}.\, \mathsf{unfolded\_circuit}(T,m) \neq \mathsf{unfolded\_faulty\_circuit}(T,m) \quad (6.3)$$

where the sequence of test input vectors T is of fixed length, and the unfolded circuits are combinational.

Universal Quantification

Since the decompositions of the unfolded circuits in Problem (6.3) are Boolean formulas, Problem (6.3) can be viewed as a quantified Boolean formula (QBF). The satisfiability problem of QB formulas (QSAT) is strictly more complex than SAT; it is PSPACE-complete [Papadimitriou, 1994]. Due to the sizes of modern VLSI circuits, already the complexity of the propositional sub-problem is enormous. Therefore, all sequential ATPG approaches I am aware of do not deal with (6.3) as a QB formula, but restrict and simplify the problem. The common principle is to eliminate the quantifier from a variable and instead mark it with a specific non-Boolean value u:

$$\text{remove} \qquad \forall m_i \in \{0,1\} \qquad \text{and add} \qquad m_i = \mathsf{u}.$$

The meaning of u is as in [Kleene, 1952, p. 334]. i. e., that of an *unknown*. As an example, Table 6.4 gives the definition of or in the resulting 3-valued logic. In this way, we reduce Problem 6.3 to

$$m = \langle\overbrace{\mathsf{u},\ldots,\mathsf{u}}^{n_{\mathrm{mem}}}\rangle,\ \mathsf{unfolded\_circuit}(T,m) \neq \mathsf{unfolded\_faulty\_circuit}(T,m), \quad (6.4)$$

which consists of a fixed number of unquantified variables.

Clearly, information is lost when eliminating quantifiers in this way: some valid tests cannot be found in the modified model. Here is an illustration.

6.2.3. Example. Consider the fragment

$$\forall M \in \{0,1\}.\ \big(\mathsf{not}(M) = N,\ \mathsf{xor}(M,N) = O\big).$$

Just considering the gates, we find that the value of O is necessarily 1, independent of the value of M. In contrast, using the 3-valued logic and $M = \mathsf{u}$ one only gets $O = \mathsf{u}$. So we cannot find a test for a stuck-at-0 fault at O, using the 3-valued approach. □

We proceed now by giving in detail several models of Problem 6.4 as a constraint satisfaction problem. We assume that the sequential circuit is given by the gate decomposition of its underlying combinational circuit. The goal is to find a test pattern $T = \langle t_1, \ldots, t_{n_{\mathrm{tp}}} \rangle$ of fixed length n_{tp}.

6.2.3 3-valued Model

We can use the 3-valued logic for a direct translation of the decompositions of both the correct and the defect circuit into separate constraints.

Each of the n_{tp} time-frames contributes the following:

Variables. For every proper input signal, there is one variable. For every output and internal signal, and for every pseudo-input and pseudo-output, there are two variables, one for the correct and one for the faulty circuit.

Domains. Proper input variables range over $\{0, 1\}$. All other variables (including the pseudo-inputs) have the domain $D_3 = \{0, 1, \mathsf{u}\}$.

Constraints.

- Every circuit gate is represented two times by the corresponding constraint in the 3-valued logic; once for the correct and once for the faulty circuit.
- In the faulty circuit, the gate with the fault at its input or output is modified by the procedure (6.1) in Section 6.1.1.

The output variables of the final time-frame are constrained to show the fault. That means, two variables from correct and faulty circuit corresponding to the same output must exist that have different Boolean values.

The n_{tp} resulting sub-CSPs for the separate combinational circuit copies, i. e. time-frames, must also be linked, by equating the corresponding pseudo-outputs and pseudo-inputs of adjacent time-frames.

Additionally, to help find short test patterns fast, state repetitions can be prevented. The corresponding constraint can be seen as an alldifferent constraint on the states. A state is given by the pair of the pseudo-output vectors (correct and faulty circuit), which represents the next circuit state.

$0/0 = 0$,	$1/0 = \mathsf{d}$,	$\mathsf{u}/0 = \mathsf{u}0$
$0/1 = \overline{\mathsf{d}}$	$1/1 = 1$,	$\mathsf{u}/1 = \mathsf{u}1$
$0/\mathsf{u} = 0\mathsf{u}$	$1/\mathsf{u} = 1\mathsf{u}$	$\mathsf{u}/\mathsf{u} = \mathsf{u}$

Figure 6.7: The elements of the 9-valued logic

$\mathsf{faultsite\_sa1}_9$									
In_{faulty}	0	1	d	$\overline{\mathsf{d}}$	u	$0\mathsf{u}$	$1\mathsf{u}$	$\mathsf{u}0$	$\mathsf{u}1$
Out_{faulty}	$\overline{\mathsf{d}}$	1	1	$\overline{\mathsf{d}}$	$\mathsf{u}1$	$\overline{\mathsf{d}}$	1	$\mathsf{u}1$	$\mathsf{u}1$

Table 6.5: Fault site constraint for the 9-valued logic

The advantage of this model is that the involved gate constraints are simple (consult Table 6.2 for or_3), and hence constraint propagation is fast. Disadvantages exist, however. Two almost equal circuit copies are dealt with in parallel. Since the input values are shared, much of the information computed and stored in the two circuit variants is the same. Furthermore, information split over two variable domains is less precise: necessary equality $\{\langle 0,0\rangle, \langle 1,1\rangle\}$, or necessary disequality $\{\langle 0,1\rangle, \langle 1,0\rangle\}$, cannot be represented as a Cartesian product $Dom_{\text{correct}} \times Dom_{\text{faulty}}$ with $Dom_{\text{correct}}, Dom_{\text{faulty}} \subseteq \{0,1\}$.

6.2.4 9-valued Model

The 4-valued logic useful for combinational test pattern generation, Section 6.2.1, can be lifted to the sequential case. The underlying principle remains the combined treatment of the signal values in correct and faulty circuit, but instead of $|\{0,1\} \times \{0,1\}| = 4$ values we deal now with $|\{0,1,\mathsf{u}\} \times \{0,1,\mathsf{u}\}| = 9$ values; listed in Fig. 6.7. The first report on the use of this 9-valued model for ATPG (but not constrained-based) is [Muth, 1976].

The definition of the gate constraints is straightforwardly based on the two combined gates. For instance, we have

$$\mathsf{or}_9 = \{\ \langle \mathsf{d},\ 0\mathsf{u},\ 1\mathsf{u}\rangle,\ \langle \mathsf{u}0,\ 0\mathsf{u},\ \mathsf{u}\rangle,\ \langle \mathsf{d},\ \mathsf{u},\ 1\mathsf{u}\rangle,\ \ldots\ \} \qquad \text{(81 tuples).}$$

Here is the CSP model based on the 9-valued logic. Each of the n_{tp} timeframes contributes the following:

Variables. For every input, output and internal signal, and for every pseudo-input and pseudo-output, there is one variable.

Domains. Proper input variables range over $\{0,1\}$. All other variables (including the pseudo-inputs) have the domain $D_9 = \{0,1,\mathsf{d},\overline{\mathsf{d}},\mathsf{u},\mathsf{u}0,\mathsf{u}1,0\mathsf{u},1\mathsf{u}\}$.

Constraints.

$0/0 = 0$,	$1/0 = \mathsf{d} = \mathsf{e}$,	$\mathsf{u}/0 = \mathsf{u}0$
$0/1 = \overline{\mathsf{d}} = \overline{\mathsf{e}}$	$1/1 = 1$,	$\mathsf{u}/1 = \mathsf{u}1$
$0/\mathsf{u} = 0\mathsf{u}$	$1/\mathsf{u} = 1\mathsf{u}$	$\mathsf{u}/\mathsf{u} = \mathsf{u}$

Figure 6.8: The elements of the 11-valued logic

- Every circuit gate is represented by the corresponding constraint in the 9-valued logic.
- The variable I_f representing the faulty line is replaced by a fresh variable O_f wherever it is used as an input variable (we include here the case that I_f is a circuit output). The variables I_f, O_f are constrained by the appropriate fault site constraint. For the stuck-at-1 fault, Table 6.5 shows this constraint.

The output variables of the final time-frame are constrained to show the fault: one variable must exist that takes the value d or $\overline{\mathsf{d}}$.

The reason for a specific fault site constraint is that we cannot inject a fault value $\mathsf{d}, \overline{\mathsf{d}}$ at *each* fault site. Only one time-frame is needed in which the fault is activated and starts the fault effect path to the final output. The fault sites at other time frames, while existent, may or may not have an effect.

The 9-valued logic clearly provides much more precise information per variable domain. Necessary equality is represented by the $\{0, 1\}$, a necessary fault effect is given by $\{\mathsf{d}, \overline{\mathsf{d}}\}$. The price is much more complex constraints: or_9 is defined by 81 tuples, in contrast to or_3 with only nine.

6.2.5 11-valued Model

Also the 6-valued logic of Section 6.2.1, used in combinational test pattern generation, can be lifted to the sequential case. So we integrate the u-value, the combined treatment of correct and faulty circuit, and fault effect distinction into required and expendable. The idea is to carry over the heuristic concerning the fault effect path (d-values versus e-values) to sequential ATPG. The resulting 11 values are listed in Fig. 6.8.

The definition of the gate constraints is straightforwardly based on the two combined gates. For instance, we have

$$\mathsf{or}_{11} = \{\ \langle \mathsf{e},\ 0\mathsf{u},\ 1\mathsf{u}\rangle,\ \langle \mathsf{d},\ \mathsf{e},\ \mathsf{d}\rangle,\ \langle \mathsf{d},\ \mathsf{u},\ 1\mathsf{u}\rangle,\ \dots\ \} \qquad \text{(93 tuples)},$$

but for example $\langle \mathsf{d}, 0\mathsf{u}, 1\mathsf{u}\rangle$ is *not* in or_{11}, since d-values must not disappear.

The CSP model follows. Each of the n_{tp} time-frames contributes the following.

Variables. For every input, output and internal signal, and for every pseudo-input and pseudo-output, there is one variable.

faultsite_sa1$_{11}$												
In_{faulty}	0	0	1	e	$\overline{\mathsf{e}}$	$\overline{\mathsf{e}}$	u	0u	0u	1u	u0	u1
Out_{faulty}	$\overline{\mathsf{d}}$	$\overline{\mathsf{e}}$	1	1	$\overline{\mathsf{d}}$	$\overline{\mathsf{e}}$	u1	$\overline{\mathsf{d}}$	$\overline{\mathsf{e}}$	1	u1	u1

Table 6.6: Fault site constraint for the 11-valued logic

Domains. Proper input variables range over $\{0, 1\}$. All other variables have the domain $D_{11} = \{0, 1, \mathsf{d}, \overline{\mathsf{d}}, \mathsf{e}, \overline{\mathsf{e}}, \mathsf{u}, \mathsf{u0}, \mathsf{u1}, \mathsf{0u}, \mathsf{1u}\}$.

Constraints.

- Every circuit gate is represented by the corresponding constraint in the 9-valued logic.
- The variable I_f representing the faulty line is replaced by a fresh variable O_f wherever it is used as an input variable (we include here the case that I_f is a circuit output). The variables I_f, O_f are constrained by the appropriate fault site constraint. Table 6.6 shows this constraint for the stuck-at-1 fault.
- Fan-out points are modelled by constraints, not by multiple variable occurrences. The constraint is the straightforward extension of the fan-out constraint from the 6-valued logic, Table 6.3, by the 5 new values using u, which are simply copied to both outputs.

As usual, the output variables of the final time-frame are constrained to show the fault, this time we can require that *exactly* one variable must exist that takes the value d or $\overline{\mathsf{d}}$.

6.3 Implementation

We have implemented sequential ATPG as described in the preceding sections. Fig 6.9 gives the model-independent algorithm schema. The CSP $\mathcal{P}$ is constructed according to the respective model of 3, 9, or 11 values. $\mathcal{P}$ is solved by first activating the fault in a chosen time-frame, and then searching through the variable assignments of first the 'future' and then the 'past' time-frames. In the 'future' time-frames, variables along a potential fault effect path (where correct line and faulty line can still differ) are preferred by the search heuristic. In case of failure, a different fault-activating time-frame is tried. Search otherwise takes place per time-frame. Within a time-frame, the first-fail heuristic is applied, that is, from all variables one with a smallest domain is chosen and instantiated [Haralick and Elliott, 1980].

The implementation of this control algorithm is realised in ECL^{i}PS^e [Wallace et al., 1997] and consists of about 5000 lines of source code.

$\boxed{\textsf{ATPG}}$: $\textsf{circuit} \longmapsto$ test pattern set

$TP := \emptyset$
for each circuit line L and fault type `saX` **do**
$TP := TP \cup \textsf{ATPG\_fault}(\textsf{circuit}, L/\texttt{saX}, 1)$
end
return TP

$\boxed{\textsf{ATPG\_fault}}$: circuit, fault, $n_{\text{tp}} \longmapsto$ test pattern

$\mathcal{P}$:= sequence of n_{tp} time-frames with inputs tp as a CSP
if solving $\mathcal{P}$ is successful **then return** $\{tp\}$
else return ATPG_fault(circuit, fault, $n_{\text{tp}} + 1$)

Figure 6.9: Algorithm schema for sequential ATPG

6.3.1 Constraint Propagation

Almost all constraints in the presented ATPG constraint models represent gates, which are operators in the chosen logic (of 3, 9, or 11 values). All these constraints are defined by an explicit set of solutions. The constraint propagation of these constraints can therefore be conducted by membership rules.

We are interested in membership rule sets sufficient to establish the local consistency notion of generalised arc-consistency (GAC). Additionally we look at sets of *equality rules* [Apt and Monfroy, 2001]. Recall that an equality rule is a specific membership rule whose condition constraints $x \in S$ are such that S is either a singleton set $S = \{a\}$ or the complete variable base domain $S = D$. In other words, either x is instantiated or its current domain is irrelevant. This feature leads to faster testing of the rule condition. On the other hand, equality rules enforce a local consistency weaker than GAC.

In the remainder of this section we always mean GAC-enforcing membership rules when we write 'membership rules'.

Automatic Generation of Propagation Rules

For the generation of membership rules, we proceed as explained in Section 5.4.4. For equality rules, we use the program of [Apt and Monfroy, 2001] (kindly provided to us by the authors). In both cases, we then remove redundant rules as described in Section 4.3.1. Finally, for each constraint, the rule scheduler R of Section 3.4.2 is instantiated with the corresponding rule set. The final result is thus an ECL^{i}PS^e source code file that provides handling and propagation of the associated constraint.

We report in the following some details of the specific rule generation processes.

3-valued logic. We require rules for the constraints and_3, or_3, and_3, or_3, and not_3 (see 6.4 for the definition of or_3). Their definitions are small: the binary operators have 9 solutions and not_3 has just 3 rules. This leads to few rules: 13 membership rules and 16 equality rules are generated for the binary operators, and 6 membership rules and as many equality rules for not_3. Generation took place in not more than one second per rule set.

9-valued logic. The number of solutions for the constraints representing binary operators is here 81; the corresponding membership rule sets are of size 385, while only 134 equality rules are obtained. Rule generation took about one minute for the membership rules, and less than a second for equality rules. Additionally to the logical operator constraints, we also have the fault-site constraints; see, e. g., Table 6.5. The 9 solutions lead to 7 membership and 13 equality rules (in both the cases `sa0`, `sa1`).

11-valued logic. The constraints for the binary operators have 93 solutions, 393 membership rules, and 153 equality rules. Membership rule generation took about 4 minutes, and redundancy removal another 10 minutes. Beside the logical constraints, there are the fault-site constraints and the fan-out constraint. The latter has 13 solutions, 33 membership rules, and 39 equality rules.

6.3.2 Empirical Evaluation

We verify the applicability of our approach to sequential ATPG by way of benchmarking. Table 6.7 shows the test generation times for some circuits. Test patterns for every possible fault are generated, i. e., every line and every stuck-at fault (0,1) is considered separately. We set a maximum test pattern length of 5, and a time limit of 20 seconds per fault.

Circuit S is our example circuit, Fig. 6.3, taken from [Muth, 1976]. The remaining circuits are ISCAS'89 circuits. The ISCAS'89 benchmark is a publicly available set of specifications of 31 sequential circuits [Brglez et al., 1989].

We give in Table 6.7 the results for the three models, and for equality rules and GAC-enforcing membership rules. The circuit features are the number of inputs, outputs, logic gates + not-gates, and memory elements. (The number of logic gates counts the original gates that sometimes have more than 2 inputs.)

We can observe that generally equality rules are the better choice for the smaller circuits. An explanation for this may be that search and propagation in the induced small CSPs leads often directly to variable instantiations as opposed to mere domain reductions, and equality rules cannot cause constraint propagation from non-singleton domains.

Next, we notice that the 3-valued logic performs best most of the time. This can be explained by the simpler constraints (and smaller rule sets), compared with

	3-valued		**9-valued**		**11-valued**		**Circuit features**			
	MEM	EQU	MEM	EQU	MEM	EQU	I	O	$G+N$	M
S	0.03	0.02	0.47	0.05	0.79	0.08	3	1	$6+2$	1
s27	0.07	0.12	1.34	0.24	1.75	0.98	4	1	$8+2$	3
s298	1288	3589	2591	4584	2825	4924	3	6	$75+44$	14
s344	681	6964	2458	7036	3090	7007	9	11	$101+59$	15
s641	3550	10831	2186	12171	2814	13028	35	24	$107+272$	19

Table 6.7: Results for benchmark circuits (times in seconds)

the other models. The advantage of the 3-valued model decreases as the circuits become more complex, however. So the more precise signal representation that is possible in the higher-valued logics may become more relevant then.

Finally, the extra domain knowledge that is built into the 11-valued model by the additional e-values essentially does not pay off in our examples.

6.4 Final Remarks

We showed the viability of a constraint-based approach to ATPG for sequential circuits. We started with a systematic treatment of constraint-based models for combinational ATPG, and extended them subsequently to the sequential case by our method of quantifier elimination and the unfolding technique.

Of particular interest in our approach is how we implement the constraint propagation. Once the logics and correspondingly the constraints are defined, which is part of modelling, the generation of constraint propagation rules (GAC-enforcing membership and equality rules) is completely automatic.

In [Simonis, 1989] as well as in [Hentenryck et al., 1992], the constraint propagation is described in rule form. All rules are manually constructed from the constraint definitions. While this is feasible for small constraints and few rules (the logics have 4 and 6 values), that is clearly not the case when one deals with larger logics leading to hundreds of rules. Furthermore, we consider redundancy in rule sets, an issue not mentioned in [Simonis, 1989] nor in [Hentenryck et al., 1992].

A constraint-based approach to combinational ATPG with the 6-valued model is discussed in [Abdennadher and Rigotti, 2004] in the context of automatic generation of constraint propagation rules. We show such rules in Section 4.4.1.

The automatic generation of constraint propagation rules is also applicable to several of the specific multi-valued logics presented in [Azevedo, 2003], where topics related to combinational ATPG, including differential diagnosis and test pattern optimisation, are extensively studied from a constraint-based perspective.

Chapter 7

Constraint-Based Modal Satisfiability Checking

7.1 Introduction

Relational structures, such as trees, graphs, transition systems, often provide a natural way to model evolving systems. One may have to deal with such relational structures for a variety of reasons, e.g. to evaluate queries, to check requirements, or to make implicit information explicit. Modal and modal-like logics such as *temporal logic* and *description logic* provide a convenient and computationally well-behaved formalism to represent such reasoning [Blackburn et al., 2001, Halpern et al., 2001].

A wide range of initiatives aimed at developing and refining algorithms for solving the satisfiability problem of basic modal logic has taken place in the past decade, driven by an increased computational usage of modal-like logics. These efforts have resulted in a series of implementations. Some implement special-purpose algorithms for modal logic, others exploit existing tools for example for propositional or first-order logic through some encoding. We follow here the second approach. The modal satisfiability problem is modelled and solved as a sequence of constraint satisfaction problems.

Specifically, we stratify a modal satisfiability problem (PSPACE-complete), into layers of constraint satisfaction problems (individually NP-complete). We refine the model substantially by exploiting the restricted syntactic nature of modal problems and the expressive power of constraints — in particular, we use not only the Boolean values. The resulting constraint satisfaction problems can be solved by a moderately expressive constraint solving system. Most of the constraints in our model are simple ones (e.g., at_most_one) for which propagation algorithms are part of many current constraint solving systems. Hence we can solve the modal satisfiability problem essentially by controlling a standard constraint solver. We demonstrate this point by an implementation in the constraint solving platform ECL^{i}PS^e system [Wallace et al., 1997]. While it cannot yet fully compete with today's highly optimised modal provers, our experimental evaluations suggest that the approach is very promising in general. Moreover, it is excellent in some cases.

The main contributions of our work derive from our modelling of modal satisfiability problems: modal formulas are translated into layers of finite constraint problems that have *non-Boolean* domains, i. e. with further values than 0 or 1, together with *appropriate constraints* to reason about these values. We show that our modelling has a number of benefits over existing encodings of modal formulas into sets of propositions. For instance, the extended domains together with appropriate constraints give us a *better control* over the modal search procedure. They allow us to set strategies on the variables to split on in the constraint solver in a compact way. Specifically, by means of appropriate constraints for our model, we can obtain satisfying partial Boolean assignments instead of total assignments.

Background

We have a broad view of what modal logic is. In this view, modal logic encompasses such formalisms as temporal logic, description logic, feature logic, dynamic logic. . . . While originating from philosophy, for the past three decades the main innovations in the area of modal logic have come from computer science and artificial intelligence. The modern, computationally motivated view of modal logic is one that takes modal logics to be expressive, yet computationally well-behaved fragments of first-order or second-order logic. Other computer science influences on modal logic include the introduction of many new formalisms, new algorithms for deciding reasoning tasks, and, overall, a strong focus on the interplay between expressive power and computational complexity. We now give some examples of modern computational uses of modal-like logics.

We start with a brief look at the use of modal-like logics in the area of formal specification and verification; a comprehensive introduction is provided by [Huth and Ryan, 1999]. Requirements such as "the system is *always* dead-lock free" or "the system *eventually* waits for a signal" can be compactly expressed in the basic modal logic by augmenting propositional logic with two operators: $\Box$ for the guarded universal quantifier over states (commonly read as *always*, meaning "in all the reachable states"), and $\Diamond$ for its existential counterpart (commonly read as *eventually*, meaning "in some reachable state"). If we formalise the statement "the system is dead-lock free" with the proposition $\mathsf{s\_free}$, and "the system waits for a signal" with $\mathsf{s\_wait}$, then the two requirements mentioned above correspond to the modal formulas $\Box\mathsf{s\_free}$ and $\Diamond\mathsf{s\_wait}$, respectively.

Multi-modal logics are popular in the agent-based community, see e. g. [Rao and Georgeff, 1998]. Each agent is endowed with beliefs and knowledge, and with goals that it needs to meet. The beliefs and knowledge can be expressed by means of multi-modal operators: $\Box^{\mathrm{b}}_{A}$ for "agent A believes" and $\Diamond^{\mathrm{b}}_{B}$

for "agent B disbelieves"; $\Box^{\mathrm{k}}_B$ for "agent B knows" and $\Diamond^{\mathrm{k}}_A$ for "agent A ignores". More complex modal formulas involving *until* operators or path quantifiers are used to reason about plans of agents, in particular to express and verify specifications on plans, see e. g. [Bacchus and Kabanza, 2000], or extended goals; see [Pistore and Traverso, 2001] for example.

Description logics are a family of modal-like logics that are used to represent knowledge in a highly structured form, using (mostly) unary and binary relations on a domain of objects [Baader et al., 2003]. Knowledge is organised in terminological information (capturing definitions and structural aspects of the relations) and assertional information (capturing facts about objects in the domain being modelled). For instance, an object satisfies $\Diamond_R A$ if it is R-related to some object satisfying A. In the area of description logic, a range of algorithms for a wide variety of reasoning tasks has been developed.

Many more areas exist in which modal-like logics are currently being used, including semi-structured data [Marx, 2004], game theory [Harrenstein et al., 2002], or mobile systems [Cardelli and Gordon, 2000]. What all of these computational applications of modal-like logics have in common is that they use relational structures of one kind or another to model a problem or domain of interest, and that a modal-like logic is used to reason about these structures. For many of the applications mentioned here, *modal satisfiability checking* — does a given modal formula have a model (an assignment to the variables) — is the appropriate reasoning task.

Related work

The past decade has seen a wide range of initiatives aimed at developing, refining, and optimising algorithms for solving the satisfiability problem of basic modal logic. Some of these implement special purpose algorithms for modal logic, such as `DLP` [Patel-Schneider, 2002], `FaCT` [Horrocks, 2002], `RACER` [Haarslev and Möller, 2002], `*SAT` [Tacchella, 1999], while others exploit existing tools or provers for first-order logic, e. g. `MSPASS` [MSPASS, 2001], or propositional logic, for instance `KSAT` [Giunchiglia and Sebastiani, 2000], `KBDD` [Pan et al., 2002], through some encoding. In this work we follow the latter approach: we propose to model and solve modal satisfiability problems as constraint problems.

The starting-points of our work are [Giunchiglia and Sebastiani, 2000] and [Areces et al., 2000]. In [Giunchiglia and Sebastiani, 2000], modal formulas are modelled as sets of propositions (i. e. Boolean formulas) stratified into layers. The propositions are processed starting from the top layer in a depth-first left-most way, and solved by a propositional solver.

We add a refinement that builds on ideas due to [Areces et al., 2000]. There, an improvement of an existing encoding of modal formulas into first-order formulas was introduced. It enables one to re-use existing first-order theorem provers for deciding modal satisfiability, and, at the same time, to inform the prover about the restricted syntactic nature of first-order translations of modal formulas. This technique resulted in a significant improvement in performance.

We build on this insight. We improve on the modelling of modal formulas with respect to [Giunchiglia and Sebastiani, 2000] so as to be able to make efficient use of existing constraint solvers to decide modal satisfiability. Specifically, modal formulas are translated into layers of finite constraint satisfaction problems that have domains with non-Boolean values together with appropriate constraints.

While the well-known DPLL algorithm can also return partial Boolean assignments for propositions, there are two key add-ons of our modelling in this respect. First, the use of extended domains and constraints allows more control over the partial assignments returned by the constraint solver than unit propagation allows in DPLL. Second, we can run any constraint solver on top of our modelling to obtain partial assignments. It is by modelling that we obtain partial assignments, and not by modifying existing constraint solvers nor by choosing a specialised solver.

7.2 Propositional Formulas as Constraint Satisfaction Problems

We begin by making a relation between propositional logic formulas and constraints.

7.2.1 Propositions

A propositional formula ϕ is a term constructed from propositional (Boolean) variables (i. e., variables with domain $\{0, 1\}$) and the propositional connectives such as $\neg, \wedge, \vee, \rightarrow$, with the usual interpretation. A positive literal is a propositional variable, a negative literal is a negated variable. When a Boolean-valued assignment μ satisfies a propositional formula ϕ, we write $\mu \models \phi$. We denote by $CNF(\phi)$ the result of ordering the propositional variables in ϕ and transforming ϕ into *conjunctive normal form (CNF)*: i. e., a conjunction of disjunctions of literals without repeated occurrences. A *clause* of ϕ is a conjunct of $CNF(\phi)$.

Propositions as CSPs

It is straightforward to transform a propositional formula into a CSP that is satisfiable exactly if the formula is. First the formula is transformed to CNF. Then each resulting clause is viewed as a constraint. For example, the CNF formula

$$\phi = (\neg x \vee y \vee z) \;\wedge\; (x \vee \neg y) \tag{7.1}$$

corresponds to the CSP

$$\mathcal{P}_\phi = \langle C_1(x, y, z), C_2(x, y) \,;\, x, y, z \in \{0, 1\}\rangle$$

in which the constraint C_1 forbids the assignment $\{x \mapsto 1, y \mapsto 0, z \mapsto 0\}$; and the constraint C_2 disallows the assignment $\{x \mapsto 0, y \mapsto 1\}$. The relation of propositional formulas and CSPs is studied extensively in [Walsh, 2000].

7.2.2 Partial Assignments

A constraint solver presented with a formula encoded as a CSP will return a total assignment to the propositional variables. In contrast, we are here interested in *partial* propositional assignments to the variables. For example, the assignment

$$\{x \mapsto 1, z \mapsto 1\}$$

satisfies the formula ϕ in (7.1) but is silent about the variable y.

One way to get such partial but satisfying propositional assignments, without modifying the underlying constraint solver, is to encode the propositional formula into a CSP with an extra value beside the Boolean 0 and 1. We use the additional value "u" to mark those propositional variables that are not required in an assignment satisfying the encoded propositional formula. So we encode proposition ϕ from (7.1) into a CSP $\mathcal{P}_\phi$ in such a way that

$$\{x \mapsto 1, y \mapsto \mathsf{u}, z \mapsto 1\}$$

is a solution to $\mathcal{P}_\phi$, from which in turn we obtain the desired partial satisfying assignment above.

Let us give a precise definition of the new encoding. We assume from now on an implicit total order on the propositional variables; it lets us ignore the order of occurrence of variables in a clause, (e. g., we do not distinguish $x \vee y$ and $y \vee x$).

7.2.1. Definition. Given a propositional formula ψ, we denote by $\mathit{CSP}(\psi)$ the CSP $\langle \mathcal{C}; X \in \mathcal{D}\rangle$ associated with it. It is defined as follows:

1. X is the ordered sequence of propositional variables occurring in ψ.
2. A domain $D_i = \{0, 1, \mathsf{u}\}$ is associated with each x_i in X.
3. For each clause θ in $\mathit{CNF}(\psi)$, a constraint C_θ exists that is on the variables $Y = y_1, \ldots, y_m$ occurring in θ. A tuple $d = d_1, \ldots, d_m$ from the product of the domains of Y satisfies C_θ if some variable y_k exists such that
   - $d[y_k] = 1$ if y_k occurs positively in θ,
   - $d[y_k] = 0$ if y_k occurs negatively in θ.

□

We give no further requirements in this definition on how constraints are represented and implemented; on purpose, as such detail is not necessary for the theoretical results concerning the modal satisfiability solver. Nevertheless, some modelling choices and implementation details are discussed in Section 7.4 below.

The modelling of propositional formulas as in Definition 7.2.1 allows us to make any complete solver for finite CSPs return a partial Boolean assignment that satisfies a propositional formula ψ if ψ is satisfiable.

We denote by $\mu|_{Bool}$ the Boolean sub-assignment of μ, that is, the set $\{\, x \mapsto b \mid (x \mapsto b) \in \mu \text{ and } b \in \{0,1\} \,\}$.

7.2.2. Theorem. *Consider a propositional formula ψ and let X be its ordered sequence of variables.*

1. *a total assignment μ for $CSP(\psi)$ satisfies $CSP(\psi)$ if and only if $\mu|_{Bool}$ satisfies ψ;*
2. *ψ is satisfiable if and only if a complete constraint solver returns a total assignment μ for $CSP(\psi)$ such that $\mu|_{Bool}$ satisfies ψ.*

Proof. First notice that a proposition and its CNF are equivalent: a Boolean assignment satisfies one exactly if it satisfies the other. Item 1 follows from this fact, Definition 7.2.1, and the following property of CNF formulas: a partial Boolean assignment μ satisfies $CNF(\psi)$ exactly if, for each clause ϕ of $CNF(\psi)$, μ assigns 1 to at least one positive literal in ϕ, or 0 to at least one negative literal in ϕ. Item 2 follows from the former. □

It is sufficient that each domain D_i of $CSP(\psi)$ contains the Boolean values 0 and 1 for the above result to hold. Thus, one could have values other than u (and 0 and 1) in the CSP modelling to mark some variables with different 'levels of relevance' for deciding the satisfiability of a formula. The choice for just one non-Boolean value as in Definition 7.2.1 suffices for our purposes.

7.3 Modal Formulas as Layers of Constraint Satisfaction Problems

In this section we recall the basics of modal logic and provide a link between solving modal satisfiability and CSPs.

7.3.1 Modal Formulas as Layers of Propositions

We refer to [Blackburn et al., 2001] for extensive details on modal logic. To simplify matters, we will focus on the basic mono-modal logic $\mathcal{K}$, although our approach can easily be generalised to a multi-modal version.

Modal formulas. $\mathcal{K}$-formulas are defined as follows. Let P be a finite set of propositional variables. Then $\mathcal{K}$-formulas over P are produced by the rule

$$\phi \quad ::= \quad p \mid \neg\phi \mid \phi \wedge \phi \mid \Box\phi$$

where $p \in P$. The formula $\Diamond p$ abbreviates $\neg\Box\neg p$, and the other Boolean connectives are defined in terms of $\neg, \wedge$ as usual. A formula of the form $\Box\phi$ is called a *box formula*.

Here and in the remainder, we always assume that P is implicitly ordered to avoid modal formulas only differing in the order of their propositional variables. Furthermore, standard propositional simplifications such as the removal of double occurrences of $\neg$ are implicitly performed on modal formulas.

Modal Layers and Propositional Approximations

The satisfiability procedure for $\mathcal{K}$-formulas that we develop in this section revolves around two main ideas:

- the stratification of a modal formula into layers of formulas of decreasing *modal depth*;
- the *approximation* and solving of such formulas as propositions.

Let us make these ideas precise, starting with the former. In words, the modal depth of a formula measures the maximum number of nested boxes. Formally, the *modal depth* of a formula ϕ is defined by

$$\begin{aligned}
\mathit{modal_depth}(p) &= 0, \\
\mathit{modal_depth}(\neg\phi) &= \mathit{modal_depth}(\phi), \\
\mathit{modal_depth}(\phi_1 \wedge \phi_2) &= \max(\mathit{modal_depth}(\phi_1), \mathit{modal_depth}(\phi_2)), \\
\mathit{modal_depth}(\Box\phi) &= \mathit{modal_depth}(\phi) + 1.
\end{aligned}$$

For instance, we have $\mathit{modal\_depth}(\Diamond\Box p \vee \Box\neg q) = 2$.

Testing if a modal formula is satisfiable involves stratifying it into layers of subformulas (or Boolean combinations of these) of decreasing modal depth. At each such layer, modal formulas are approximated and solved as propositions. Formally, the *propositional approximation* $\mathit{Prop}(\phi)$ of a formula ϕ is the propositional formula defined inductively by

$$\begin{aligned}
\mathit{Prop}(p) &= p \\
\mathit{Prop}(\neg\phi) &= \neg\mathit{Prop}(\phi) \\
\mathit{Prop}(\phi_1 \wedge \phi_2) &= \mathit{Prop}(\phi_1) \wedge \mathit{Prop}(\phi_2) \\
\mathit{Prop}(\Box\phi) &= x[\Box\phi].
\end{aligned}$$

We denote here by $x[\Box\phi]$ a fresh propositional variable that is associated with one specific occurrence of $\Box\phi$. Different occurrences of $\Box\phi$ lead to different variables which are distinguished by an index.

For instance, the formula $\phi = p \wedge \Box q \vee \neg\Box q$ is approximated by the propositional formula $Prop(\phi) = p \wedge x_1[\Box q] \vee \neg x_2[\Box q]$. The variables of ϕ are $\{p, q\}$ while the variables of $Prop(\phi)$ are $\{p,\ x_1[\Box q],\ x_2[\Box q]\}$.

7.3.2 $\mathcal{K}$-satisfiability and the k_sat Schema

We formalise here $\mathcal{K}$-satisfiability, and present the general algorithm schema k_sat for deciding the satisfiability of $\mathcal{K}$-formulas. It is given in Fig. 7.1. The k_sat algorithm schema is the base of the KSAT algorithm of [Giunchiglia and Sebastiani, 2000].

$\mathcal{K}$-satisfiability

At this point we have to make a choice between a standard characterisation of the semantics of $\mathcal{K}$-formulas, and one closer to the semantics of the solving algorithm. We choose for the latter as this allows us to come more quickly and concisely to the matter of interest. For the standard characterisation we refer to [Blackburn et al., 2001], for example.

7.3.1. Definition. The $\mathcal{K}$-formula ϕ is $\mathcal{K}$-*satisfiable* if

- a Boolean assignment μ exists that satisfies $Prop(\phi)$,
- for every variable $x[\Box\lambda]$ of $Prop(\phi)$ such that $x[\Box\lambda] \mapsto 0$ is in μ, the $\mathcal{K}$-formula

$$\neg\lambda \ \wedge\ \bigwedge\{\,\theta \mid \mu(x[\Box\theta]) = 1\,\}$$

is $\mathcal{K}$-satisfiable.

□

We use here, and from now on, a simpler notation for the conjunction of the elements of a set, namely we write

$$\bigwedge S \quad \text{to abbreviate} \quad \bigwedge_{e \in S}$$

and do so analogously for a disjunction.

$\boxed{\mathsf{sat}}$: propositional formula $\longmapsto$ satisfying assignment or failure

If the formula is propositionally satisfiable, then return a Boolean assignment. Return alternatives on backtracking.

$\boxed{\mathsf{k\_sat}}$: modal formula ψ $\longmapsto$ succeeds if ψ satisfiable

$\mu := \mathsf{sat}(\mathit{Prop}(\psi))$ // *create a choice point*

$B^+ := \{\, \theta \mid x[\Box\theta] \mapsto 1 \text{ is in } \mu \,\}$
$B^- := \{\, \lambda \mid x[\Box\lambda] \mapsto 0 \text{ is in } \mu \,\}$

$\Theta := \bigwedge B^+$
for each $\lambda \in B^-$ **do**
    $\mathsf{ksat}(\Theta \wedge \neg\lambda)$ // *backtrack if this fails*
end

Figure 7.1: The k_sat algorithm schema.

The k_sat Algorithm Schema

In the k_sat schema, provided in Fig. 7.1, the sat procedure determines the satisfiability of the propositional approximation of ϕ by returning a Boolean assignment μ as in Definition 7.3.1. Alternative satisfying assignments are generated upon backtracking. If there is no alternative assignment, then the call to k_sat fails at this level and backtracking takes place, unless it is the top level in which case it is reported that the formula is unsatisfiable.

In this way, the modal search space gets stratified into modal formulas of decreasing modal depth and explored depth-first. A variable of the form $x[\Box\lambda]$ to which μ assigns 0 means that we must "open the box" and check λ against all the formulas θ that come with variables of the form $x[\Box\theta]$ to which μ assigns 1. Exactly one proposition is so created and tested satisfiable.

7.3.2. THEOREM. *In the* k_sat *algorithm schema given in Figure 7.1, if* sat *is a complete solver for Boolean formulas, then* k_sat *is a decision procedure for* $\mathcal{K}$*-satisfiability.*

PROOF. Correctness and completeness of k_sat is entailed by the characterisation of $\mathcal{K}$-satisfiability in Definition 7.3.1. k_sat terminates since the modal depth and the number of propositional variables of a modal formula are bounded. $\Box$

7.3.3 The KCSP Algorithm

We now devise a modal decision procedure based on the k_sat schema, parameterised by a constraint solver as the underlying propositional solver sat. We first provide the intuition by an example.

7.3.3. Example. Consider the modal formula

$$\phi \;=\; \neg\Box(p \vee \bot) \;\wedge\; (\Box r \vee \Box p).$$

(The symbol $\bot$ abbreviates the 'always false' formula and could be defined as $p' \wedge \neg p'$ for some arbitrary $p' \in P$.) The propositional approximation $Prop(\phi)$ of ϕ, can be turned into a CSP according to Def. 7.2.1. We obtain

- three variables $x[\Box(p \vee \bot)]$, $x[\Box r]$, and $x[\Box p]$, each with domain $\{0, 1, \mathsf{u}\}$,
- two constraints,
  - one for $\Box(p \vee \bot)$, forcing the assignment 0 to $x[\Box p \vee \bot]$,
  - one for $(\Box r \vee \Box p)$, requiring 1 to be assigned to $x[\Box r]$ or $x[\Box p]$.

Assigning the value u to a variable means not committing to any decision concerning its Boolean values, 0 and 1. This CSP is given to the constraint solver, which may return the assignment

$$\begin{aligned}\mu_1 = \{\quad & x[\Box(p \vee \bot)] \mapsto 0,\\ & x[\Box r] \mapsto \mathsf{u},\\ & x[\Box p] \mapsto 1 \quad \}.\end{aligned}$$

Then, for all the variables $x[\Box \ldots]$ to which μ_1 assigns 1 (in this case only $x[\Box p]$), the formulas within the scope of $\Box$ are joined in a conjunction Θ. So we have

$$\Theta = p. \tag{UT}$$

Then all the box variables to which μ_1 assigns 0 are considered, in this case only $x[\Box(p \vee \bot)]$. Thus, the formula $p \vee \bot$ is negated, simplified (translated in CNF when needed), and we obtain

$$\neg\lambda = \neg p. \tag{ET}$$

The conjunction $\Theta \wedge \neg\lambda$ is given to the sat solver. In this case, the clause passed on is $p \wedge \neg p$. It is translated into a new CSP and given to the constraint solver, which results in failure due to its inconsistency.

On the subsequent backtracking, we may obtain the alternative assignment

$$\begin{aligned}\mu_2 = \{\quad & x[\Box(p \vee \bot)] \mapsto 0,\\ & x[\Box r] \mapsto 1,\\ & x[\Box p] \mapsto \mathsf{u} \quad \}.\end{aligned}$$

This assignment leads to $\Theta = r$, and thus to the formula $r \wedge \neg p$ which is successfully tested satisfiable. In turn, satisfiability of ϕ is concluded. $\Box$

Notice the key points about (UT) and (ET): we only consider the box variables $x[\Box\ldots]$ to which a Boolean value, 0 or 1, is assigned. The box variables to which u is assigned are disregarded, safely so because of Theorem 7.2.2. We show in the following section that the availability of values other than 0 and 1 has a number of advantages.

7.3.4. DEFINITION. The KCSP *algorithm* is defined as follows. In the $\mathsf{k\_sat}$ schema, the sat function whose input is a formula ϕ, is instantiated with a complete solver for finite CSPs whose input is $\mathit{CSP}(\mathit{Prop}(\phi))$. $\Box$

We can state the following by Theorems 7.2.2 and 7.3.2.

7.3.5. COROLLARY. KCSP *is a decision procedure for $\mathcal{K}$-satisfiability.* $\Box$

7.4 Constraint-Based Modelling

In this section we discuss the constraints into which we translate a modal formula. We begin with a base modelling, and proceed to an improved modelling that possesses some desirable properties.

When modelling a problem as a CSP for solving it in a propagation & search-based solver, one generally has two options for the user-defined constraints. Either one implements a custom-built constraint propagation procedure for such a constraint, or one rewrites it into constraints for which propagation algorithms are available. Obviously the latter approach is preferable, all other things being equal. We follow it here.

7.4.1 Base Modelling

The input to KCSP is a formula in conjunctive normal form. We translate it into a CSP clause-wise, each clause contributing one constraint

Aspect 1: Clauses as Constraints

We distinguish four disjoint sets of variables in a clause: propositional variables and variables representing box formulas, and both subdivided according to polarity. We denote these sets P^+, P^-, B^+, B^-, resp. Therefore, a clause can be written as

$$\begin{array}{llllllllll} \bigvee\{ & p & \mid & p \in P^+\} & \vee & \bigvee\{ & \neg p & \mid & p \in P^-\} & \vee \\ \bigvee\{ & x[\Box\phi] & \mid & x[\Box\phi] \in B^+\} & \vee & \bigvee\{ & \neg x[\Box\phi] & \mid & x[\Box\phi] \in B^-\} & \end{array}$$

Such a clause is now viewed as a constraint on the variables in the four sets,

$$\mathsf{clause\_constraint}(P^+, P^-, B^+, B^-).$$

It holds if *at least one* variable in the set $P^+ \cup B^+$ is assigned a 1, or one variable in $P^- \cup B^-$ is assigned a 0; recall Definition 7.2.1. We decompose this constraint now in terms of a simpler constraint $\mathsf{at\_least\_one}$, which is defined on a set of variables and parameterised by a constant, and which requires the latter to occur in the variable set. This constraint, or a related one, is available in many constraint programming systems. The constraint library of ECL^{i}PS^e contains a predefined constraint with the meaning of $\mathsf{at\_most\_one}$, which can be employed to imitate $\mathsf{at\_least\_one}$.

Thus, as the first step to obtain a method for propagation of clause constraints, we reformulate $\mathsf{clause\_constraint}$ as the disjunctive constraint

$$\mathsf{at\_least\_one}(P^+ \cup B^+, 1) \quad \vee \quad \mathsf{at\_least\_one}(P^- \cup B^-, 0).$$

Aspect 2: Disjunctions as Conjunctions

Propagating disjunctive constraints, if supported at all, is generally difficult for constraint solvers. It is therefore preferable to avoid them when modelling, and in our situation that can be done rather elegantly. The disjunctive clause constraint is transformed into a conjunction with the help of a single auxiliary link variable. We obtain

$$\mathsf{at\_least\_one}(P^+ \cup B^+ \cup \{\ell\}, 1) \quad \wedge \quad \mathsf{at\_least\_one}(P^- \cup B^- \cup \{\ell\}, 0).$$

The link variable $\ell \in \{0, 1\}$ selects implicitly which of the two constraints must hold. For example, observe that $\ell = 0$ selects the constraint on the left: it forces $\mathsf{at\_least\_one}(P^+ \cup B^+, 1)$ and satisfies $\mathsf{at\_least\_one}(P^- \cup B^- \cup \{0\}, 0)$.

It is useful to remark that this way of rewriting the clause constraint does not hinder constraint propagation in the sense that if GAC is established on the two $\mathsf{at\_least\_one}$ constraints separately then the clause constraint is GAC; by Lemma 2.1.7 and since the two conjuncts share no variables except ℓ.

7.4.2 Advanced Modelling

We extend the base constraint model so as to generate partial assignments, avoid CNF conversion, and take into account multiple occurrences of box formulas.

Aspect 3: Partial Assignments by Constraints

While any solution of the CSP induced by a formula at some layer satisfies the formula at that layer, it is useful to obtain satisfying partial Boolean assignments that mark as irrelevant as many box formulas in this layer as possible. This causes fewer subformulas to enter the propositions generated for the subsequent layer. We use the extra value u to mark irrelevant variables.

7.4.1. EXAMPLE. Take the clause $p \vee \neg q \vee \Box r \vee \neg\Box s$. The corresponding clause constraint is on the four groups of variables $P^+ = \{p\}$, $P^- = \{q\}$, $B^+ = \{x[\Box r]\}$, and $B^- = \{x[\Box s]\}$, that is (in simpler notation),

$$\mathsf{clause\_constraint}(p, q, x[\Box r], x[\Box s]).$$

Some solutions to this constraint are *better than others*. For instance,

$$\begin{aligned}
\mu_1 &= \{\underline{p \mapsto 1}, \underline{q \mapsto 0}, x[\Box r] \mapsto \mathsf{u}, x[\Box s] \mapsto \mathsf{u}\},\\
\mu_2 &= \{\underline{p \mapsto 1}, q \mapsto 1, x[\Box r] \mapsto \mathsf{u}, x[\Box s] \mapsto \mathsf{u}\},\\
\mu_3 &= \{p \mapsto 0, \underline{q \mapsto 0}, x[\Box r] \mapsto \mathsf{u}, x[\Box s] \mapsto \mathsf{u}\},\\
\mu_4 &= \{p \mapsto 0, q \mapsto 1, \underline{x[\Box r] \mapsto 1}, x[\Box s] \mapsto \mathsf{u}\},\\
\mu_5 &= \{p \mapsto 0, q \mapsto 1, x[\Box r] \mapsto \mathsf{u}, \underline{x[\Box s] \mapsto 0}\},
\end{aligned}$$

are 'good' solutions since they assign a Boolean value to as few variables representing box formulas as possible. Note that, within these five assignments, μ_1, μ_2, μ_3 are preferable, but since p, q may also be constrained by other clauses, we must admit μ_5, μ_6 as well.

In contrast, for example

$$\begin{aligned}
\mu_6 &= \{p \mapsto 1, q \mapsto 0, x[\Box r] \mapsto 0, x[\Box s] \mapsto \mathsf{u}\},\\
\mu_7 &= \{p \mapsto 0, q \mapsto 1, x[\Box r] \mapsto 1, x[\Box s] \mapsto 0\}.
\end{aligned}$$

are undesirable, as box formulas are unnecessarily marked to be considered in the next layer. □

We formalise this observation.

7.4.2. FACT. Consider a CSP containing a clause constraint C on the propositional variables $P = P^+ \cup P^-$ and the variables representing box formulas $B = B^+ \cup B^-$. Suppose μ is a partial assignment that is *not* on the variables $P \cup B$.

- The assignment $\mu' = \mu \cup \{p \mapsto 1\} \cup \{\, x_\Box \mapsto \mathsf{u} \mid x_\Box \in B \,\}$, where $p \in P^+$, satisfies C. (Analogously for $p \in P^-$.)
- The assignment $\mu' = \mu \cup \{x_\Box \mapsto 1\} \cup \{\, x'_\Box \mapsto \mathsf{u} \mid x'_\Box \in B - \{x_\Box\} \,\}$, where $x_\Box \in B^+$, satisfies C. (Analogously for $x_\Box \in B^-$.)

If μ can be extended to a total assignment satisfying the CSP than also μ' can be so extended. □

Note that the variables in B are constrained only by C, and recall Theorem 7.2.2.

In other words, if satisfying the propositional part of a clause suffices to satisfy the whole clause, then all box formulas in it can be marked irrelevant. Otherwise, all box formulas except one can be marked irrelevant.

Let us transfer this observation into a clause constraint model. First, we rewrite the base model so as to

- separate the groups of variables (in propositional and box variables),
- and convert the resulting disjunctions into conjunctions, again with the help of extra linking variables.

Next, we replace the $\mathsf{at\_least\_one}$ constraint for variables representing box formulas by an $\mathsf{exactly\_one}$ constraint. This simple constraint is commonly available as well; ECL^{i}PS^e offers the more general $\mathsf{occurrences}$ constraint, which forces a certain number of variables in a set to be assigned to a specific value. We obtain

$$\begin{aligned}
&\mathsf{clause_constraint}(P^+, P^-, B^+, B^-) \;=\\
&\qquad \mathsf{at_least_one}(P^+ \cup \{\ell_P^+\}, 1) \quad \wedge \quad \mathsf{exactly_one}(B^+ \cup \{\ell_B^+\}, 1) \quad \wedge\\
&\qquad \mathsf{at_least_one}(P^- \cup \{\ell_P^-\}, 0) \quad \wedge \quad \mathsf{exactly_one}(B^- \cup \{\ell_B^-\}, 0) \quad \wedge\\
&\qquad \mathsf{clause_link}(\ell_P^+, \ell_P^-, \ell_B^+, \ell_B^-)
\end{aligned}$$

The variable domains are: $p \in \{0, 1\}$ for $p \in P$, next $x_\Box \in \{1, \mathsf{u}\}$ for $x_\Box \in B^+$, and $x_\Box \in \{0, \mathsf{u}\}$ for $x_\Box \in B^-$.

The essential four linking variables are constrained as in the following logical formula or the equivalent table.

$$\begin{aligned}
&\mathsf{clause_link}(\ell_P^+, \ell_P^-, \ell_B^+, \ell_B^-) \;=\\
&\quad (\ \ell_P^+ = 1 \wedge \ell_P^- = 0\ ) \ \leftrightarrow\ (\ \ell_B^- = \mathsf{u}\ \vee\ \ell_B^+ = \mathsf{u}\ )\\
&\quad \wedge\\
&\quad \ell_B^+ = 1\ \vee\ \ell_B^- = 0
\end{aligned}$$

ℓ_P^+	ℓ_P^-	ℓ_B^+	ℓ_B^-
1	0	1	u
1	0	u	0
0	0	1	0
0	1	1	0
1	1	1	0

Observe that the 5 tuples in the table correspond to the situations that we wish to permit — the clause is satisfied by either a positive or a negative box formula (but not both at the same time) or a positive or a negative propositional variable (maybe both at the same time). Compare also with $\mu_{1..6}$ in Example 7.4.1.

ECL^{i}PS^e can accept the linking constraint in logical-operator form, in which case it is internally rewritten into several arithmetic constraints. Alternatively, we can compile the defining table into a set of membership rules, for instance by the method described in Section 5.4.4. Examples for the generated rules are

$$\begin{aligned}
\ell_P^+ = 0 \quad &\rightarrow \quad \ell_B^+ \neq \mathsf{u},\ \ell_B^- \neq \mathsf{u},\\
\ell_P^- = 0,\ \ell_B^+ = 1,\ \ell_B^- = 0 \quad &\rightarrow \quad \ell_P^+ \neq 1.
\end{aligned}$$

We found in our experiments that the linking constraint, among all constraints, is the one whose propagation is executed most often. Hence propagating it efficiently is particularly relevant. Using the generated membership rules and the corresponding scheduler (Chapter 3.4.2) proved to be the fastest way of propagating the linking constraint of several methods we tested.

Aspect 4: A Negated-CNF Constraint

Except for the initial input formula to KCSP which is in conjunctive normal form, the input to an intermediate call to the sat function of KCSP (see the algorithm in Figure 7.1) has the form $\Theta \wedge \neg\lambda$ where both Θ and λ are formulas in CNF. A naive transformation of $\neg\lambda$ into CNF will result in an exponential increase in the size of the formula. We deal with this problem by treating $\neg\lambda$ as a constraint. The following holds.

7.4.3. FACT. The constraint $\neg\lambda$ is satisfiable if and only if λ (which is a conjunction of clauses) has at least one unsatisfiable clause. $\square$

We formulate the constraint corresponding to $\neg\lambda$ consequently as a disjunctive constraint, each disjunct standing for a negated clause. This disjunctive constraint is converted into a conjunction using a set $L = \{\ell_1, \ldots, \ell_m\}$ of linking variables, one for each of the m disjuncts. Every ℓ_i ranges over $\{1, \mathsf{u}\}$. The case $\ell_i = 1$ means the ith disjunct holds, i. e., the ith clause in λ is unsatisfied. Instead of imposing $\mathsf{at\_least\_one}(L, 1)$ to select one disjunct to hold, however, we require $\mathsf{exactly\_one}(L, 1)$, in line with our goal of obtaining minimal partial Boolean assignments. $\ell_i = \mathsf{u}$ means the ith disjunct (clause) is irrelevant. We then forcibly mark all box formulas by u, but ignore the propositional variables.

The definition of the negated-clause constraint on the variables in P^+, P^-, B^+, and B^-, and the linking variable ℓ_i in logical form is

$$\mathsf{negated\_clause}(P^+, P^-, B^+, B^-, \ell_i) \ =$$

$$\begin{array}{llll} \ell_i = 1 & \rightarrow & (\ \forall p \in P^+ .\ p = 0 \quad \wedge \quad \forall p \in P^- .\ p = 1\ ) & (NC_1) \\ \wedge & & & \\ \ell_i = 1 & \leftrightarrow & (\ \forall b \in B^+ .\ b = 0 \quad \wedge \quad \forall b \in B^- .\ b = 1\ ) & (NC_{2a}) \\ \wedge & & & \\ \ell_i = \mathsf{u} & \leftrightarrow & (\ \forall b \in B^+ .\ b = \mathsf{u} \quad \wedge \quad \forall b \in B^- .\ b = \mathsf{u}\ ). & (NC_{2b}) \end{array}$$

Constraint Propagation

Let us sketch the process of developing a constraint propagation procedure for negated_clause as an example for a specialised constraint. Notice first that negated_clause consists of conjuncts NC_1 and NC_2. We discuss them separately.

From NC_1, we can immediately read off the membership rule

$$\ell_i = 1 \quad \rightarrow \quad p \neq 1$$

where $p \in P^+$ or $p \in P^-$. The domain of p is $\{0, 1\}$.

A membership rule as r_1 that is atomic (has just equalities in the condition) corresponds to a non-solution of the associated constraint; recall Note 5.3.4. If we wish to obtain all correct atomic membership rules for a constraint, we can construct each atomic membership rule from each non-solution, and combine the rules. See the treatment of this issue in the context of incremental rule generation, in particular Section 5.4.3.

The (partial) non-solution $\langle 1, 1\rangle$ for $\langle \ell, p\rangle$ in NC_1, which underlies the rule r_1, admits exactly one other rule, namely

$$p = 1 \quad \rightarrow \quad \ell_i \neq 1$$

No other correct membership rule is possible for conjunct NC_1.

Propagating the conjunct NC_2 is substantially simplified by the constraint propagation rule

$$NC_2 \quad \rightarrow \quad b_1 = b_2$$

for all $b_1, b_2 \in B^+$, or $b_1, b_2 \in B^-$. In presence of equality constraint propagation, we need thus only deal with two representative variables $b_k^+ \in B^+$, $b_k^- \in B^-$, which are constrained with the linking variable by

$$\langle \ell_i, b_k^+, b_k^-\rangle \in \{\, \langle 1, 0, 1\rangle,\ \langle \mathsf{u}, \mathsf{u}, \mathsf{u}\rangle \,\}.$$

This restriction is just a simple extensionally defined constraint. Membership rules for it can be generated as described in Section 5.4.4, for example.

The thus developed propagation rules reflect the propagation algorithm for the negated_clause constraint as implemented in KCSP. It establishes generalised arc-consistency.

Aspect 5: A Constraint for Factoring

In our base model, we have treated and constrained the ith occurrence of a box formula $\Box\phi$ as a distinct propositional variable $x_i[\Box\phi]$. For instance, the two occurrences of $\Box p$ in the formula $\Box p \wedge \neg\Box p$ are treated as the two distinct propositional variables $x_1[\Box p]$ and $x_2[\Box p]$ in our base model.

We consider here the case that a box formula occurs several times, in several clauses, in any polarity. We prevent assigning conflicting values to different occurrences, by a suitable constraint.

Let us collect in the set $B_{\Box\phi}$ all variables $x_i[\Box\phi]$ representing the formula $\Box\phi$ in the entire CSP. Recall that their domain is $\{0, 1, \mathsf{u}\}$. We state as a constraint that

$$\forall x_1, x_2 \in B_{\Box\phi}.\ \big(\neg(\ x_1 = 1 \wedge x_2 = 0\ )\ \wedge\ \neg(\ x_1 = 0 \wedge x_2 = 1\ )\big).$$

To see the effect, suppose there is a pair $x_1, x_2 \in B_{\Box\phi}$ such that $x_1 \mapsto 1$, $x_2 \mapsto 0$ in a solution to the CSP without this factoring constraint. This means we obtain both $\Box\phi \mapsto 1$ and $\Box\phi \mapsto 0$ in the assignment returned, which results in an unsatisfiable proposition $\phi \wedge \neg\phi \wedge \ldots$ being generated. The factoring constraint just detects such failures earlier. The straightforward modelling idea, namely using one unique variable for representing a box formula in all clauses ('factoring out' the formula), clashes with the assumption made for the other partial-assignment constraints that each box formula variable is unique.

Propagation rules for the factoring constraint can be derived in a similar way as for negated_clause, and lead to an implementation that establishes GAC.

7.5 Implementation and Experimental Assessment

Theoretical studies often do not provide sufficient information about the effectiveness and behaviour of complex systems such as satisfiability solvers and their optimisations. Empirical evaluations must then be used. In this section we provide an experimental comparison of our advanced modelling (Section 7.4.2) against the base model (Section 7.4.1), using the test developed in [Heuerding and Schwendimann, 1996].

We find that, no matter what other models and search strategies we commit to, we always get the best results by using constraints for partial assignments as in Section 7.4.2, Aspect 3. In the remainder of this paper, these are referred to as the *assignment-minimising*, or simply minimising, constraints. We show below how these minimising constraints allow us to better direct the modal search procedure.

We conclude this section by comparing the version of KCSP that features the advanced modelling with KSAT. The constraint solver that we use as the sat function in KCSP is a conventional one, based on search with chronological backtracking and constraint propagation. The propagation algorithms are specialised for their respective constraints and enforce generalised arc-consistency on them, as discussed in Section 7.4 above.

7.5.1 Test Environment

State of the Art

In the area of propositional satisfiability checking there is a large and rapidly expanding body of experimental knowledge; see, e. g. [Gent et al., 2000]. In contrast, empirical aspects of modal satisfiability checking have only recently drawn the attention of researchers. We now have a number of test sets, some of which have been evaluated extensively [Baader et al., 1992, Heuerding and Schwendimann, 1996, Giunchiglia and Sebastiani, 2000, Hustadt and Schmidt, 1997, Horrocks et al., 2000]. In addition, we also have a clear set of guidelines for performing empirical testing in the setting of modal logic [Heuerding and Schwendimann, 1996, Horrocks et al., 2000].

Currently, there are three main test methodologies for modal satisfiability solvers, one based on hand-crafted formulas, the other two based on randomly generated problems.

To understand on what kinds of problems a particular prover does or does not do well, it helps to work with test formulas whose meaning can (to some extent) be understood. For this reason we opted to carry out our tests using the Heuerding and Schwendimann (HS) test set [Heuerding and Schwendimann, 1996], which was used at the TANCS '98 comparison of systems for non-classical logics [TANCS, 2000].

The HS Test Set

The HS test set consists of several classes of formulas for $\mathcal{K}$ and other modal logics that we do not consider here. Some problem classes for $\mathcal{K}$ are based on the pigeon-hole principle (*ph*) and a two-colouring problem on polygons (*poly*). We consider the classes *branch*, *d*4, *dum*, *grz*, *lin*, *path*, *ph*, *poly*, *t*4.

Each class is generated from a parameterised logical formula. This formula is either a $\mathcal{K}$-theorem, which is thus provable, or only $\mathcal{K}$-satisfiable, which is not provable. So the generated class contains either *p*rovable formulas or *n*on-provable formulas, and is labelled accordingly by a suffix *_p* or *_n*.

Some of these parameterised formulas are made harder by hiding their structure or adding extra material. The parameter allows for the creation of modal formulas in the same class but of differing difficulty. Specifically, the formulas in a class are constructed in such a way that the difficulty of proving them should be exponential in the parameter. It is hoped that this kind of increase in difficulty makes differences in the speed of the machines used to run the benchmarks less significant.

Benchmark Methodology

The benchmark methodology is to test formulas from each class starting with the easiest instance (controlled by the parameter), until the provability status of a formula can not be correctly determined within 100 CPU seconds. The test result of this class is the parameter of the largest formula that can be solved within this time limit. The parameter ranges from 1 (easiest) to 21 (hardest).

Implementation

We implemented the KCSP algorithm as a prototype in the constraint programming system ECL^{i}PS^e. The HS formulas are negated, reduced to CNF and translated into the format of KCSP.

We add heuristics to KCSP with minimising constraints in an attempt to reduce the depth of the KCSP search tree. The value u is preferred for box formulas, and among them for positively occurring ones. Furthermore, the instantiation

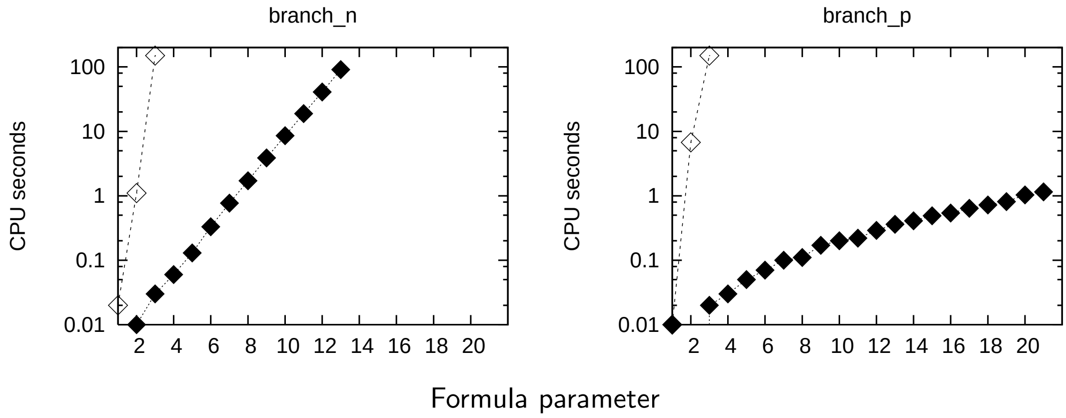

Figure 7.2: KCSP with (♦) and without (◊) assignment minimisation.

ordering of variables representing box formulas is along their increasing modal depth, e. g., $x[\Box p]$ is instantiated before $x[\Box\Box p]$.

7.5.2 Assessment

In this subsection we evaluate the contributions of the various aspects of our advanced modelling.

Aspect 3: Partial assignments by constraints. Do minimising constraints make a difference in practice? To address this question, we focus here on the *branch* formulas in the HS test set. This class of formulas is specifically relevant for automated modal theorem proving. Its non-provable variant *branch_n* is recognised as the hardest class of "truly modal formulas" for today's modal theorem provers [Horrocks et al., 2000]. They are the so-called *Halpern and Moses branching formulas* that "have an exponentially large counter-model but no disjunction [...] and systems that will try to store the entire model at once will find these formulae even more difficult" [Horrocks et al., 2000].

Figure 7.2 plots the run times of KCSP on *branch* formulas, with and without minimising constraints. There is clearly a difference. KCSP with assignment-minimising constraints manages to solve 13 instances of *branch_n* and all 21 of *branch_p* (in less than 2 seconds). Without minimisation, in both cases only 2 instances are solved.

To understand the reasons for the good performance of KCSP with minimising constraints, consider *branch_p*(3), unsolved by KCSP with total assignments. In KCSP with minimising constraints, there are two choices for box formulas at the root layer and none at the subsequent layer of modal formulas obtained by "opening" a box. This small factor results in a modal search tree of just two branches.

In contrast, with total assignments there are 6 extra box formulas at the root layer. The implied extra branching factor is $2^6 = 64$ at the root of the modal search tree only. All 6 box formulas are always carried over to subsequent layers, positively or negatively, adding to the work to be done there.

More generally, the superiority of KCSP with minimising constraints can be explained as follows: *the tree-like model that the solver implicitly attempts to construct while trying to satisfy a formula is kept as small as possible by the minimising constraints*. In this sense, constraints allow us better control over the modal search than, for instance, unit propagation allows for in DPLL.

Notice finally that the results of KCSP with minimising constraints on the *branch* class are competitive on the this class with the best optimised modal theorem provers such as `*SAT` and `DLP`.

Aspect 4: Negated-CNF constraint. In all the HS formula classes, using constraints in place of CNF conversions leaves equal or increases the number of decided formulas. Avoiding CNF conversion by means of negated-CNF constraints has a substantial effect for example in the case of *ph_n*(4), an instance of the pigeon-hole problem, which is solved in a few seconds. In contrast, by requiring CNF conversion (even with minimising constraints), the execution of KCSP is terminated pre-emptively due to memory exhaustion.

However, the CNF conversion at the root level remains necessary, and prevents formulas *ph*(k) with $k > 4$ in KCSP for lack of memory.

Aspect 5: Factoring constraint. This constraint avoids simple contradictory occurrences of a formula in the layers subsequent to the current one. We remark that this consideration of multiple occurrences of a subformula does not always provide a globally minimal number of box formulas with a Boolean value. Nevertheless, it is beneficial for formulas with the same variables hidden and repeated inside boxes. It proved useful in the cases *grz*, *d*4, *dum_p*, *path_p*, *t*4*_p*. In the remaining cases the contribution of factoring with constraints is insignificant, except for *path_n* where searching for candidate formulas to be factored slightly slowed down the search.

Formula simplifications. As a preprocess to KCSP, the top-level input formula can be simplified to a logically equivalent formula. We use standard simplification rules for propositional formulas, at all layers, in a bottom-up fashion. In the same way, some modal equivalences such as $\neg\Box\top \equiv \bot$ are used.

Simplification in KCSP plays a relevant role in the case of *lin* formulas. Without simplifications and minimising constraints, KCSP takes longer than 5 minutes to return an answer for *lin_n*(3). With simplifications and minimising constraints, the runtime is reduced to less than 0.4 seconds. By also adding factoring, KCSP solves the most difficult formula of *lin_n* in 0.06 sec, that of *lin_p* in 0.01 sec.

	branch		*d4*		*dum*		*grz*		*lin*		*path*		*ph*		*poly*		*t4*	
	n	p	n	p	n	p	n	p	n	p	n	p	n	p	n	p	n	p
KSAT	8	8	5	8	>	11	>	17	3	>	8	4	5	5	12	13	18	10
KCSP	13	>	6	9	19	12	>	13	>	>	11	4	4	4	16	10	7	10
KCSP/sp	*11*	>	6	*8*	*17*	*11*	>	*10*	>	>	*9*	4	4	4	16	*9*	*6*	*8*

Table 7.1: Results on the HS Benchmark

7.5.3 Results and a Comparison

We compare the performances of KSAT and KCSP on the HS test set in Table 7.1. Each column lists a formula class and the number of the most difficult formula decided within 100 CPU seconds per prover. We write > when all 21 formulas in the test set are solved within this time limit.

First row: KSAT. The results for KSAT are taken from [Horrocks et al., 2000]. They reflect a run of a C++ implementation of KSAT with the HS test set on a 350 MHz Pentium II with 128 MB of main memory.

Second row: KCSP. We used KCSP with all advanced aspects considered; i. e., partial assignments by constraints, negated-CNF constraints, factoring constraints, and formula simplifications. In the remainder, we refer to this as KCSP. The time taken by the translator from the HS format into that of KCSP is insignificant. The worst case among those in Table 7.1 took less than 1 second (and these timings are taken into account for the table entries). We ran our experiments on a 1.2 GHz AMD Athlon Processor with 512 MB RAM, under Red Hat Linux 8 and ECL^{i}PS^e 5.5.

Third row: KCSP/sp. To account partially for the different platforms used to run KSAT and KCSP on, we scaled the measured times of KCSP by a factor 350/1200, the ratio of the respective processor speeds. The results are reported in the line KCSP/sp, and *italicized* where different from KCSP.

Result Analysis

Some interesting similarities and differences in performance between KSAT and KCSP can be observed. For some formula classes, KCSP clearly outperforms KSAT, for some it is the other way round, and for others the differences do not seem to be significant.

For instance, KCSP performs better in the case of *lin* and *branch* formulas. As pointed out in Subsection 7.5.2, *branch_n* is the hardest “truly modal test class” for current modal provers, and KCSP with partial assignments performs well on this class. Now, similar to KCSP, KSAT features partial assignments in that its underlying propositional solver is DPLL-based. The differences in performance are then due to our modelling. In more general terms:

- Extended domains and constraints allows for *better control* over the partial assignments to be returned by the adopted constraint solver than unit propagation allows for in DPLL.

- Constraints allow a *compact representation* of certain requirements such as that of reducing the number of variables representing box formulas to which a Boolean value is assigned.

The models that KCSP (implicitly) tries to generate when attempting to satisfy a formula remain very small. In the case of *branch*, searching for partial assignments with minimising constraints yields other benefits *per se*: the smaller the number of box formulas to which a Boolean value is assigned at the current layer, the smaller the number of propositions in the subsequent layer. In this way, fewer choice points and therefore fewer search tree branches are created. Thereby, the addition of constraints to limit the number of box formulas to reason on, while still exploring fully the purely propositional search space, seems to be a useful idea on the *branch* class.

In the cases of *grz* and *t*4, on the other hand, KSAT is superior to KCSP. KSAT features a number of optimisations (heuristics) for early modal pruning that are absent in KCSP, and these may be responsible for the better behaviour of KSAT on these classes.

We remark finally that KSAT is compiled C++ code while KCSP is interpreted ECL^{i}PS^e (Prolog) code. This makes it interesting to see that KCSP competes well with KSAT.

7.6 Final Remarks

We described here a constraint-based approach to modal satisfiability testing. The reasoning process is split into sequences of propositional problems which are solved as separate constraint satisfaction problems. We showed the feasibility of our ideas by discussing an implementation and benchmark results.

Taking advantage of the expressiveness that constraint modelling affords, we integrate *control knowledge* when translating the propositions into CSPs. In particular, we observed that partial solutions to the propositional subproblems suffice and that some solutions are preferable to others. By using an extra non-Boolean domain value and additional constraints, we obtain preferred partial solutions *by modelling* instead of by modifying the solver. Such partial propositional assignments reduce substantially the branching factor in the tree-model that our solver implicitly tries to construct. The resulting reduction in search time is significant, as we found empirically.

The generated CSPs use several standard constraints, for which current constraint programming systems provide propagation algorithms. For some specialised constraints we implemented specific propagation algorithms that establish generalised arc-consistency. The underlying propagation mechanisms are derived elegantly as constraint propagation rules.

We conclude by pointing out some possible and interesting extensions.

The current modelling of propositional formulas is clause-based, assuming either CNF or negated-CNF format. A direct mapping to constraints of formulas in *any* format, i. e., using any Boolean operators, is certainly feasible, and would remove the initial CNF conversion required now. Achieving small partial assignments in this situation appears to be more involved, however.

Simple chronological backtracking to traverse the propositional (and thus also the modal) search tree is probably not the optimal choice. Efficiency can be expected to increase by learning, that is, by remembering previously failed sub-propositions (nogood recording, intelligent backtracking), and also successfully solved sub-problems (lemma caching).

Finally, we mention many-valued modal logics [Fitting, 1992]. These logics allow for propositional variables to have further values than the Boolean 0 and 1. Our approach to modal logics via constraint satisfaction appears to be particularly suitable to be naturally extended to deal with finitely-valued modal logics.

Chapter 8

Array Constraint Propagation

8.1 Introduction

Many problems can be modelled advantageously using look-up functionality: associate each item in a group of items with a unique identifier, or index, and make items directly accessible by their respective index. In mathematics, indices on variables are ubiquitous, and functions are used to uniquely map arguments to values. In programming languages, the corresponding construct is usually called *array*. In an imperative language such as `C`, we might define an array of integer variables by `integer a[3]`, or an array of constants by `a[] = {5, 7, 9}`; we can then access the element at position `i` by writing `a[i]`.

In such languages, the condition for these look-up expressions to be valid is that the index is known when the expression is evaluated. It is in the spirit of constraint programming to relax this restriction. We view x and y as variables constrained by the equality $x = a[y]$ which involves an array a.

A corresponding binary constraint named element was originally developed within the CHIP system, one of the earliest constraint programming systems, [Dincbas et al., 1988]. element proved to be very useful in modelling; many problems (scheduling, resource allocation, etc.) formulated as CSPs make use of it, and most contemporary constraint programming systems provide it now. Sometimes it is generalised so as to allow the one-dimensional array to consist of variables instead of constants. Array constraints and element are examples of so-called 'global' constraints [Beldiceanu and Contejean, 1994, Beldiceanu, 2000a].

Another point motivating the study of array constraints lies in the on-going development in constraint programming research to lift the notion of a constrained variable from the conventional numeric or simple finite-domain variable to higher-structured objects, such as vectors, sets [Gervet, 1997], multisets [Walsh, 2003]. Arrays connect to the notion of a *function variable* [Hnich, 2003]. In this view, an array mapping the indices in $\mathcal{I}$ to variables ranging over $\mathcal{A}$ is itself a single variable whose domain is the set of functions from $\mathcal{I}$ to $\mathcal{A}$.

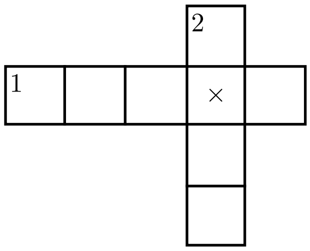

Figure 8.1: Crossing entries in a crossword puzzle

Let us demonstrate the use of arrays in modelling.

8.1.1. EXAMPLE. We formulate the problem of crossword puzzle generation using array constraints [Beacham et al., 2001]. Given a list of words and an empty crossword puzzle grid, the task is to fill the horizontal and vertical entries in the grid with words of appropriate length such that crossing entries agree on the letter at the crossing position. Figure 8.1 shows such a crossing.

We view the entries as variables w_i. Their domain is the respective set of words of appropriate size, i. e., the domain of w_1 in the figure is the set of 5-letter words.

We use a constant two-dimensional array *letter* to associate words with their letters. The first index denotes the word, the second denotes the position of a letter in that word, e. g., $\mathit{letter}[\mathsf{sail}, 2] = \mathsf{a}$. Every crossing of two entries contributes an array constraint. For example,

$$\mathit{letter}[w_1, 4] = \mathit{letter}[w_2, 2]$$

captures the crossing of Fig. 8.1.

Solely establishing GAC on the array constraints solves some instances of the crossword problem without any search; see [Hentenryck, 1989, p. 140], in which an equivalent special constraint for crossing entries is used. □

8.1.2. EXAMPLE. In Chapter 9, we discuss qualitative spatial reasoning using an array-based model. In this approach, we map tuples of objects to their spatial relation. For example, the relative orientation of point triples is represented as a three-dimensional array *OrRel* indexed by points p_i. The set of qualitative relations $\{\mathsf{between}, \mathsf{behind}, \mathsf{in\_front}, \mathsf{left}, \ldots\}$ is the domain of the array elements. We specify by $\mathit{OrRel}[a, b, c] = \mathsf{in\_front}$ that the continuation of the directed line $\overrightarrow{ab}$ passes through the point c. The constraint

$$\mathit{OrRel}[\mathsf{forward}_A, \mathit{defender}_B, \mathsf{goal}_B] = \mathsf{in\_front},$$
$$\mathit{defender}_B \in \mathit{Team}_B$$

in a football context formalises the suboptimal situation for a forward player of team A who is in possession of the ball that some player of team B prevents a direct goal shot. □

We study here constraint propagation for array constraints. Arrays can be multidimensional and they can consist of variables, the indices in an array expression can be variables, and the array expression is equated with a variable. We consider propagation establishing generalised arc-consistency and bounds-consistency. We also discuss nested array expressions. Furthermore, we examine a method to transform a multidimensional array constraint into an equivalent one-dimensional array constraint and an auxiliary constraint. Such a transformation is acceptable if GAC is the local consistency to be established by constraint propagation. If BC is to be established, we argue that this is not the case; propagating the original array constraint is then preferable.

8.1.1 Arrays

An array is a representation of a total function. Given a Cartesian product $\mathcal{I} = \mathcal{I}_1 \times \cdots \times \mathcal{I}_n$ and a set $\mathcal{A}$ for the function domain and range, respectively, an array a is a set of atomic mappings that satisfies

$$\text{for every } b \in \mathcal{I} \text{ some } e \in \mathcal{A} \text{ exists such that } (b \mapsto e) \in a.$$

We use conventional array notation and write

$$a[b] = e \qquad \text{if} \qquad (b \mapsto e) \in a.$$

The length n of $\mathcal{I} = \mathcal{I}_1 \times \cdots \times \mathcal{I}_n$ is the dimensionality of the array. We assume that all $\mathcal{I}_i$ are finite.

8.1.2 Array Constraints

We use arrays in array expressions $a[b]$ and simple array equations $e = a[b]$, where $b \in \mathcal{I}, e \in \mathcal{A}$. We lift array equations to array constraints of the form

$$x = a[y_1, \ldots, y_n]$$

by allowing variables instead of constants, as follows:

- result variable x with domain $D_x = \mathcal{A}$,
- index variables $y_1, \ldots, y_n = y$ with domains $D_{y_i} = \mathcal{I}_i$,
- array variables $a[b]$ for $b \in \mathcal{I}$ with domains $D_{a[b]} = \mathcal{A}$.

So such an array constraint is a constraint on the sequence of variables

$$X = x,\ y_1, \ldots, y_n,\ \langle a[b] \mid b \in \mathcal{I} \rangle. \tag{8.1}$$

It is of arity $1 + n + \prod_{i=1}^{n} |\mathcal{I}_i|$, and therefore highly non-binary.

We assume from now on that all the variables in the sequence X are pair-wise different. We say that a is an array of constants if all $\langle a[b] \mid b \in \mathcal{I} \rangle$ are constants, otherwise we call it an array of variables.

8.2 Constraint Propagation

8.2.1 Propagation Rules for Generalised Arc-Consistency

Simple Array Constraints

We derive GAC-establishing constraint propagation rules for array constraints from the generic rule of Fact 2.2.3. So we are interested in all correct rules of the form

$$C(x_1, \ldots, x_n) \ \to \ x_i \neq e$$

for an array constraint C. Correctness follows from $e \notin C[x_i]$.

The variables in an array constraint $x = a[y_1, \ldots, y_n]$ split in three groups, see Statement (8.1). We examine the correctness condition separately for a representative of each group.

Variable x**.** We require $e \notin C[x]$, or equivalently $e \neq a[y_1, \ldots, y_n]$. That is the case exactly if

$$\nexists b \in D_{y_1} \times \cdots \times D_{y_n}.\ e \in D_{a[b]} \tag{8.2}$$

holds for the domains.

Variable y_k**.** The correctness condition is $b_k \notin C[y_k]$. We find in this case

$$\begin{aligned} &D_{y|k} = D_{y_1} \times \cdots \times D_{y_{k-1}} \times \{b_k\} \times D_{y_{k+1}} \times \cdots \times D_{y_n}, \\ &\nexists b \in D_{y|k}.\ \exists e \in D_x \cap D_{a[b]}. \end{aligned} \tag{8.3}$$

Variable $a[b_1, \ldots, b_n]$**.** We need a circumstance in which $e \notin C[\,a[b_1, \ldots, b_n]\,]$. That is only the case once the index is fixed to $(b_1, \ldots, b_n)$; then, all other variables $a[b'_1, \ldots, b'_n]$ are unconstrained. We have thus

$$\{(b_1, \ldots, b_n)\} = D_{y_1} \times \cdots \times D_{y_n} \quad \wedge \quad e \notin D_x. \tag{8.4}$$

We now instantiate the generic GAC-establishing rule for each variable type and obtain the following three rules:

$$x = a[y_1, \ldots, y_n] \quad \to \quad x \neq e \qquad \text{if (8.2),} \qquad (arr\_gac_x)$$

$$x = a[y_1, \ldots, y_n] \quad \to \quad y_k \neq b_k \qquad \text{if (8.3),} \qquad (arr\_gac_y)$$

$$x = a[y_1, \ldots, y_n] \quad \to \quad a[b_1, \ldots, b_n] \neq e \qquad \text{if (8.4).} \qquad (arr\_gac_a)$$

8.2.1. Theorem. *The rules* (arr_gac_x)*,* (arr_gac_y)*,* (arr_gac_a) *establish generalised arc-consistency on the array constraint* $x = a[y_1, \ldots, y_n]$*.*

PROOF. Fact 2.2.3, and the preceding derivations of the respective correctness conditions. □

Pair-wise Variable Difference Requirement

It is indeed necessary to restrict variables to occur just once. Consider the array $xor = \{\langle 0,0\rangle \mapsto 0, \langle 0,1\rangle \mapsto 1, \langle 1,0\rangle \mapsto 1, \langle 1,1\rangle \mapsto 0\}$ and the constraint $x = xor[y,y]$ with $x \in \{0\}, y \in \{0,1\}$. It is inconsistent but stable under the rules (arr_gac_x), (arr_gac_y), (arr_gac_a).

Compound Array Constraints

We have only admitted array constraints in the simple form $x = a[y_1, \dots, y_n]$ so far. It can sometimes be easier, however, to use several arrays in one constraint, such as in $a_1[y_1, \dots, y_n] = a_2[z_1, \dots, z_m]$ or in the nested expression $x = a_3[a_1[y_1, \dots, y_p], a_2[z_1, \dots, z_q]]$.

Establishing GAC on such array constraints is generally hard if variables are used in multiple places. If variables occur just once then the compound expressions can simply be decomposed, using fresh auxiliary variables. Lemma 2.1.7 states that GAC on the constraints of the decomposition corresponds to GAC on the compound constraint.

So, for example, the constraint $letter[w_1, 4] = letter[w_2, 2]$ from the crossword example 8.1.1 can be decomposed into the two constraints $letter[w_1, 4] = \ell_{1,2}$ and $letter[w_2, 2] = \ell_{1,2}$ without affecting propagation.

Domain Reduction vs. Constraint Transformation

As instances of the generic GAC-establishing rule in Fact 2.2.3, the rules (arr_gac_x), (arr_gac_y), (arr_gac_a) are domain reduction rules by type. In presence of GAC-establishing constraint propagation rules or algorithms for basic constant/variable equality constraints $v_1 = v_2$, we can replace the domain reduction rule (arr_gac_a) by a constraint propagation rule that does not reduce domains but imposes the entailed equality constraint. Such equality constraints are generally provided in constraint logic programming systems, which implemented them through unification extended with domain intersection.

So we extract just

$$\{(b_1, \dots, b_n)\} = D_{y_1} \times \dots \times D_{y_n} \tag{8.5}$$

from correctness condition (8.4), and state the rule

$$x = a[y_1, \dots, y_n] \quad \rightarrow \quad x = a[b_1, \dots, b_n] \qquad \text{if (8.5).} \qquad (arr\_gac_{a,=})$$

Propagation of the new equality constraint $x = a[b_1, \dots, b_n]$ reduces then the domains of the variables x and $a[b_1, \dots, b_n]$ in the same way as (arr_gac_x), (arr_gac_a).

8.2.2. NOTE. *In presence of constraint propagation mechanisms for variable equality constraints, the rules* (arr_gac_x), (arr_gac_y), ($arr\_gac_{a,=}$) *establish GAC on the array constraint* $x = a[y_1, \ldots, y_n]$.

Moreover, observe that, as long as the domains $D_x, D_{a[b_1,\ldots,b_n]}$ are non-empty, also the rule (arr_gac_y) is redundant (more precisely: it has no correct instances). So, once ($arr\_gac_{a,=}$) has fired, the original constraint $x = a[y_1, \ldots, y_n]$ and all its propagation rules can be eliminated from the constraint solver.

8.2.2 Propagation Rules for Bounds-Consistency

Generalised arc-consistency is a strong but often also a computationally expensive local consistency. Depending on the problem, it can be more efficient to propagate less. Bounds-consistency (Def. 2.1.8) is a good candidate for a weaker local consistency notion. Recall that it checks and modifies only the domain bounds of variables. The significant implication for the representation of domains is that domains that are intervals remain intervals, which reduces the space complexity substantially.

For array constraints we obtain propagation rules for bounds-consistency from the rules establishing GAC, see Fact 2.2.3. We restrict correctness condition to domain bounds, i. e.,

$$(arr\_gac_x) \text{ in which } e \in \{\min(D_x), \max(D_x)\} \qquad (arr\_bc_x)$$

$$(arr\_gac_y) \text{ in which } b_k \in \{\min(D_{y_k}), \max(D_{y_k})\} \qquad (arr\_bc_y)$$

$$(arr\_gac_a) \text{ in which } e \in \{\min(D_{a[b_1,\ldots,b_n]}), \max(D_{a[b_1,\ldots,b_n]})\} \qquad (arr\_bc_a)$$

8.2.3. THEOREM. *The rules* (arr_bc_x), (arr_bc_y), (arr_bc_a) *establish bounds-consistency on the array constraint* $x = a[y_1, \ldots, y_n]$. □

8.2.3 From Rules to Algorithms

A naive iteration algorithm of the propagation rules establishing BC or GAC is computationally expensive, due to a repetitive access to the same variable domains in the process of verifying the correctness conditions (8.2), (8.3), (8.4).

In Figure 8.2, we give propagation algorithms for BC and GAC which implement the rule iteration process. The principle is to start with sets of values e that are candidates for removal in a body $x_i \neq e$ of an array constraint propagation rule. The algorithm core loop `array_prop` deletes all those values for which the corresponding propagation rule is incorrect. Subsequently, the remaining values can correctly be removed from the respective variable domains.

We presume that basic equality constraints are provided by the underlying constraint programming platform, and pose an equality constraint as soon as correct by condition (8.5), instead of reducing domains; see Section 8.2.1.

$\boxed{\textsf{array\_gac}}$: array constraint $\longmapsto$ equivalent GAC-reduced constraint

$\langle XU, YU_{1..n}\rangle = \textsf{array\_prop}(D_x, D_{y_1}, \ldots, D_{y_n})$
$D_x := D_x - XU$
$D_{y_i} := D_{y_i} - YU_i$, for all $i \in [1..n]$
if $\{(b_1, \ldots, b_n)\} = D_{y_1} \times \cdots \times D_{y_n}$ **then** constrain $x = a[b_1, \ldots, b_n]$

$\boxed{\textsf{array\_bc}}$: array constraint $\longmapsto$ equivalent BC-reduced constraint

let $\text{bds}(D) = \{\min(D), \max(D)\}$

$XS := \varnothing$ // *supported values*
$YS_i := \varnothing$, for all $i \in [1..n]$

repeat
  $XT := \text{bds}(D_x) \setminus XS$ // *values to be tested*
  $YT_i := \text{bds}(D_{y_1}) \setminus YS_i$, for all $i \in [1..n]$
  $\langle XU, YU_{1..n}\rangle = \textsf{array\_prop}(XT, YT_{1..n})$ // *unsupported values*
  $XS := XS \cup (XT - XU)$
  $YS_i := YS_i \cup (YT_i - YU_i)$, for all $i \in [1..n]$
  $D_x := D_x - XU$
  $D_{y_i} := D_{y_i} - YU_i$, for all $i \in [1..n]$
until $XU = \varnothing$ **and** $YU_i = \varnothing$, for all $i \in [1..n]$
if $\{(b_1, \ldots, b_n)\} = D_{y_1} \times \cdots \times D_{y_n}$ **then** constrain $x = a[b_1, \ldots, b_n]$

$\boxed{\textsf{array\_prop}}$: domain values $XU, YU_{1..n}$ $\longmapsto$ unsupported domain values

$B := D_{y_1} \times \cdots \times D_{y_n}$
while $B \neq \varnothing$ **and** $YU_k \neq \varnothing$ for some $k \in [1..n]$ **do**
  choose $(b_1, \ldots, b_n) \in B$ such that $b_k \in YU_k$ for some $k \in [1..n]$
  remove $(b_1, \ldots, b_n)$ from B
  if $D_x \cap D_{a[b_1,\ldots,b_n]} \neq \varnothing$ **then** $YU_i := YU_i \setminus \{b_i\}$, for all $i \in [1..n]$
  $XU := XU \setminus D_{a[b_1,\ldots,b_n]}$
end
while $B \neq \varnothing$ **and** $XU \neq \varnothing$ **do**
  choose and remove $(b_1, \ldots, b_n)$ from B
  $XU := XU \setminus D_{a[b_1,\ldots,b_n]}$
end
return $\langle XU, YU_{1..n}\rangle$

Figure 8.2: Propagation for array constraints

8.2.4. NOTE. *Algorithm* array_gac *establishes GAC, and algorithm* array_bc *establishes BC, on the array constraint* $x = a[b_1, \ldots, b_n]$. □

Let us examine the working of the GAC-enforcing propagation.

8.2.5. EXAMPLE. Consider $x \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}\}$ and $y_1 \in \{1, 2\}$, $y_2 \in \{1, 2, 3\}$ in the constraint $x = a[y_1, y_2]$, and let a be defined as the array of constants

$a[y_1, y_2]$	1	2	3
1	A	B	C
2	D	E	F

The constraint $x = a[y_1, y_2]$ is GAC, which array_gac verifies by calling array_prop with $XU = \{\mathsf{B}, \mathsf{C}, \mathsf{D}\}$ and $YU = (\{1, 2\}, \{1, 2, 3\})$.

Initially, the set of indices B is $\{1, 2\} \times \{1, 2, 3\}$. We iterate through B (choose statement) from lexicographically small to large indices.

1. $D_{a[1,1]} = \{\mathsf{A}\}$ is evaluated, but no changes to XU, YU result.

2. $D_{a[1,2]} = \{\mathsf{B}\}$ follows. We have $XU = \{\mathsf{C}, \mathsf{D}\}$ and $YU = (\{2\}, \{1, 3\})$.

3. $D_{a[1,3]} = \{\mathsf{C}\}$ is read, which results in $XU = \{\mathsf{D}\}$ and $YU = (\{2\}, \{1\})$.

4. Only $(2, 1)$ remains in B, so $D_{a[2,1]} = \{\mathsf{D}\}$ is looked up. $XU = \varnothing$ and $YU = (\varnothing, \varnothing)$ remain.

Only one incomplete run is needed; the indices $(2, 2), (2, 3)$ permissible by the domains of y_1, y_2 are skipped.

Observe that an alternative iteration strategy with less steps exists in this case. Suppose $(2, 1)$ had been chosen first, then only $(1, 2)$ and $(1, 3)$ could be chosen next, and $(1, 1), (2, 2), (2, 3)$ had been skipped. □

For GAC, the correctness-checking procedure array_prop iterates through all possible indices $b_1, \ldots, b_n$ in the domains of $y_1, \ldots, y_n$ in the worst case. This situation occurs, for instance, with the constraint and initial domains of Example 8.2.5 except for $x \in \{\mathsf{F}\}$, and if array_prop iterates through B from small to large indices.

In the best case, the number of iteration steps is the size of the largest domain D_{y_i}. Take Example 8.2.5, but with $x \in \{\mathsf{A}, \mathsf{E}, \mathsf{F}\}$. The algorithm iterates through $(1, 1), (2, 2), (2, 3)$ in three steps, corresponding to $|D_{y_2}| = 3$.

8.2.6. NOTE. *The number of iteration steps in* array_prop *has an upper bound of* $\mathcal{O}(d^n)$ *and a lower bound of* $\mathcal{O}(d)$*, where* d *is the size of the largest input set of values.* □

In the case of array_gac, the input sets are the complete variable domains. In the case of array_bc only the currently unchecked domain bounds are examined by one call to array_prop. If subsequently some bounds are reduced, the process needs to be repeated for the new bounds. Generally, the cost of checking the domain bounds by array_bc will be lower than the cost of checking every domain element by array_gac.

It is useful to remark that the set B in the procedure array_prop need not be represented extensionally with the resulting high space cost. Instead, a compact iterator/pointer can be maintained that marks the lexicographically next tuple.

8.3 Decomposing Multidimensional Array Constraints

Interestingly, a constraint language with one-dimensional array constraints and integer arithmetic constraints is already expressive enough to support multidimensional arrays. We discuss now a method to translate a multidimensional array constraint into a one-dimensional array constraint and an additional constraint. The interest of such a translation lies in the greater simplicity of a propagation algorithm for only one-dimensional array constraints.

We show that decomposing is an acceptable technique when the desired result of propagation is GAC. We argue that this is not the case, however, when we wish to enforce only BC.

8.3.1 Reducing the Array Dimensionality

Let the n-dimensional array a represent a total function from the Cartesian product $\mathcal{I}_1 \times \cdots \times \mathcal{I}_n = \mathcal{I}$ to the set $\mathcal{A}$. Assume that every component set is a (finite) integer interval, so

$$\mathcal{I}_i = [0 \mathrel{..} (m_i - 1)] \qquad \text{for all } i \in [1..n].$$

We can do so for our purposes without substantial loss of generality, as any finite set can be mapped to such an interval. We define a mapping f from $\mathcal{I}$ to the interval $[0 \mathrel{..} (\prod_{i=0}^{n} m_i - 1)]$, by

$$\overline{b} = f(b_1, \dots, b_n) = \sum_{i=1}^{n} \left(b_i \cdot \prod_{j=0}^{i-1} m_i \right) \tag{8.6}$$

(we define $m_0 = 1$ for convenience).

8.3.1. Example. We map car number plates labels to numbers. Let us assume that a number plate consists of a sequence of six symbols: two letters, two digits, and again two letters, e. g., RB-18-GH. A letter is taken from the Latin alphabet of 26 letters; we translate it implicitly to a number in the interval [0..25]. A number plate $p \in [0..25]^2 \times [0..9]^2 \times [0..25]^2$ with $p = (p_1, \ldots, p_6)$ can thus be mapped to a number between 0 and $26^4 \cdot 10^2 - 1$ by

$$\overline{p} = f(p) \;=\; p_1 + 26p_2 + 26^2p_3 + 10 \cdot 26^2p_4 + 10^2 \cdot 26^2p_5 + 10^2 \cdot 26^3p_6.$$

This means $f(\mathsf{R}, \mathsf{B}, 1, 8, \mathsf{G}, \mathsf{H}) = 12763599$. □

We associate the multidimensional array a with a new one-dimensional array $\overline{a}$ by

$$\overline{a} = \{\, f(b) \mapsto e \mid (b \mapsto e) \in a \,\},$$

which means that

$$\overline{a}[\, f(b_1, \ldots, b_n)\,] = a[b_1, \ldots, b_n].$$

Informally speaking, we 'linearise' a.

8.3.2 Decomposition

We deal now with array constraints (possibly on a multidimensional array) by replacing them by a new array constraint on a one-dimensional array, and an appropriate linear constraint derived from (8.6), linking the respective array indices.

8.3.2. Example. Consider the two-dimensional array a defined by

$a[y_1, y_2]$	0	1	2
0	15	16	17
1	1	2	3

We set up a new one-dimensional array $\overline{a}$ as follows:

$\overline{a}[\overline{y}]$	0	1	2	3	4	5
	15	1	16	2	17	3

We can then replace the constraint

$$x = a[y_1, y_2]$$

by the two constraints

$$x = \overline{a}[\overline{y}] \qquad \text{and} \qquad \overline{y} = y_1 + 2y_2.$$

where $\overline{y}$ is a new variable. □

8.3.3 Propagation

Generalised Arc-consistency

Enforcing GAC on the two constraints of the decomposition is equivalent to enforcing GAC on the original array constraint. This is by Lemma 2.1.7, and since the two decomposition constraints share only the new auxiliary index variable. The cost of propagation is, however, not reduced by decomposing array constraints.

8.3.3. FACT. The complexity of establishing GAC on a linear arithmetic equality constraint in n variables is in $\mathcal{O}(d^n)$, where d is the size of the largest variable domain. □

So we have the same worst-case cost for propagation via the decomposition constraints as for propagation by the array_gac algorithm.

Bounds-Consistency

If we choose bounds-consistency as the desired local consistency notion, we observe that enforcing it on the decomposition is strictly weaker than bounds-consistency on the original array constraint. The problem occurs due to the loss of information exchange between the two constraints, if only bounds-consistency is enforced.

8.3.4. EXAMPLE. Reconsider the two-dimensional array a from Example 8.3.2 and its linearised peer $\overline{a}$. Consider the variables

$$x \in [1..3], \quad y_1 \in [0..1], \quad y_2 \in [0..2].$$

The constraint $x = a[y_1, y_2]$ is clearly not bounds-consistent: for that, we must reduce the domain of y_1 to the singleton interval $[1..1]$ since $y_1 = 0$ does not occur in any solution. In contrast, the two decomposition constraints

$$x = \overline{a}[\overline{y}] \qquad \text{and} \qquad \overline{y} = y_1 + 2y_2 \qquad \text{with} \qquad \overline{y} \in [1..5]$$

are bounds-consistent. □

In conclusion, if we wish to enforce bounds-consistency on a multidimensional array constraint, we should choose the array_bc algorithm instead of decomposing the constraint.

8.4 Implementation

We implemented the algorithms array_gac and array_bc, see Fig. 8.2, in the constraint programming system ECL^{i}PS^e [Wallace et al., 1997], using its finite domain constraints library. The propagation algorithm is provided in a library together with several other array-related functions, consisting of about 600 lines of source code in total.

A specific side effect of the array propagation algorithm can be exploited in an ECL^{i}PS^e implementations. ECL^{i}PS^e controls the execution order of constraint propagation algorithms based on changes to the constrained variables, such as a domain reduction. Propagation algorithms 'watching' a variable are scheduled to execute once this variable has changed.

The array_prop procedure allows to extract useful variables to watch, namely the variables $a[b_1, \ldots, b_n]$ for which $D_x \cap D_{a[b_1,\ldots,b_n]} \neq \varnothing$. The domains of these watched variables provide support for domain values of other variables. Hence, changes to the watched variables require a repeated propagation round.

Array constraints and the implementation of propagation algorithms are reused in the Chapters 9 and 10 on qualitative reasoning.

8.5 Final Remarks

Related Work

The established precursor of array constraints is the element constraint of CHIP [Dincbas et al., 1988], now available in many constraint programming languages. It is the one-dimensional specialisation, and usually requires the array to be constant.

Algorithms for propagation in the one-dimensional case have been published, for example, [Carlson et al., 1994] describes an AKL(FD) implementation of element using indexicals [Codognet and Diaz, 1996], in which the array can consist of variables. I am not aware of a published algorithm for the multidimensional case.

Array constraints in the constraint programming language OPL [Hentenryck et al., 1999] can be multidimensional and use arrays of variables. It is unclear what form of constraint propagation takes place, but in [Hentenryck et al., 1999, p. 100] it is stated that the reduction for an index variable in a multidimensional array constraint depends on its position. We found experimentally in the OPL implementation available to us that the propagation is weaker than GAC (and BC). For all three cases treated by the rules (arr_gac_x), (arr_gac_y), (arr_gac_a), we could construct simple examples using small 2-dimensional arrays in which reduction of domains is possible but not performed by OPL Studio 3 [ILOG, 2000], see Fig. 8.3.

```
enum      Dz  { i, j };
enum      Dy  { k, l, m };
enum      Da  { p, q, r };

Da        a[Dz, Dy] = #[  i:  #[k:p, l:q, m:r]#,
                          j:  #[k:p, l:q, m:r]#   ]#;
var       Da  x;
var       Dz  z,u;
var       Dy  y,v;

solve {   v <> l;         //           OPL Studio        GAC
          a[u, v] = x;    //  x  in   { p, q, r }      { p, r }
                          //
          a[z, y] = q;    //  y  in   { k, l, m }      { l }
};
```

```
enum      Dy  { i, j, k };
enum      Da  { p, q, r };

var       Da  a[Dy];
var       Da  x;
var       Dy  y;

solve {   y = j;
          x <> q;       //             OPL Studio        GAC
          x = a[y];     //  a[j]  in  { p, q, r }      { p, r }
};
```

Figure 8.3: OPL programs exhibiting weak propagation in ILOG's OPL Studio

In [Beldiceanu, 2000b] a constraint called case is proposed that subsumes multidimensional array constraints with constant arrays. No algorithm is given. In [Hooker et al., 2000] on combining operations research techniques and constraint satisfaction methods, a continuous relaxation of element using a cutting-planes approach is studied. The element constraint there corresponds to a one-dimensional array of variables with continuous domains.

Conclusions

We studied here constraint propagation for array constraints. There is ample evidence suggesting that arrays are useful for modelling constraint satisfaction problems. Indices on objects are ubiquitous in mathematics. Arrays with multiple dimensions have long been present in programming languages. The element constraint is supported by many constraint systems.

Practical experience shows that the most advantageous notion of local consistency depends on the considered problem. Sometimes a weaker notion such as bounds-consistency may suffice, perhaps just applied in the early stages of search and later replaced by full generalised arc-consistency. We derived constraint propagation rules to achieve generalised arc-consistency and bounds-consistency, and we gave algorithms implementing the rules.

We also examined the option of decomposing a multidimensional array constraint into one with just a one-dimensional array and a linear constraint. We argued that when we wish to establish GAC on array constraints, the composed and the decomposed behave similarly with respect to runtime, while this is not the case when we require only BC. We showed that decomposing a multidimensional array constraint results in a loss of information when just BC is enforced on the sub-constraints of the decomposition. In this case, it is more appropriate to use a BC algorithm, such as the one we propose, on the original non-decomposed array constraint.

Chapter 9

Constraint-Based Qualitative Spatial Reasoning

9.1 Introduction

Qualitative reasoning was introduced in AI to abstract from the numeric quantities, such as the precise time of an event, or the concrete location or velocity of an object, and to reason instead on the level of appropriate abstractions.

In the literature two different forms of qualitative reasoning were studied. The first one is concerned with reasoning about continuous change in physical systems, monitoring streams of observations and simulating behaviours, to name a few applications. The main techniques used are qualitative differential equations, constraint propagation and discrete state graphs. For a thorough introduction see [Kuipers, 1994].

The second one aims at reasoning about contingencies such as time, space, shape, size, directions, etc. through an abstraction of the quantitative information into a finite set of qualitative relations. One then relies on complete knowledge about the interrelationship between these qualitative relations. This approach is exemplified by temporal reasoning due to [Allen, 1983], spatial reasoning introduced in [Egenhofer, 1991] and [Randell et al., 1992a], reasoning about cardinal directions (such as North, Northwest, etc), see, e. g., [Ligozat, 1998], etc.

A recent and comprehensive overview of the qualitative spatial reasoning (QSR) field is provided by [Cohn and Hazarika, 2001].

Constraint-Based Models

Qualitative spatial representation and reasoning problems lend themselves well to modelling by constraints. In the standard modelling approach, a spatial object, such as a region, is described by a variable, and the qualitative relation between spatial objects, such as the topological relation between two regions, contributes a constraint.

For many QSR calculi it is known how, in a semantical respect, global feasibility of a spatial specification corresponds to local feasibility. In the case of a fully specified topological scenario, for instance, if for all three objects the respective three binary topological relations are compositionally consistent, then the entire scenario is consistent. In the standard model with relations as constraints, this condition corresponds to path-consistency (PC, Def 2.1.10).

If the qualitative relation between two objects is not fully specified, the corresponding constraint is a disjunction of basic constraints. By establishing PC, such a disjunctive constraint is refined in view of the constraints with which it shares a variable. A combination of PC with search over the disjunctive constraints decides the consistency of indefinite scene descriptions.

We examine here an alternative constraint-based formulation of QSR. In this approach, a spatial object is a constant, and the relation between spatial objects is a variable. We call this the *relation variable* approach, in contrast to the conventional *relation constraint* approach.

Although the use of relation variables is not original, see [Tsang, 1987] where relation variables are proposed for qualitative temporal reasoning, it is mentioned very rarely in the QSR literature. This fact surprises in view of the advantages of this approach. In particular, the following important issues are tackled successfully:

Aspect integration. Space has several aspects that can be characterised qualitatively, such as topology, size, shape, relative and absolute orientation. These aspects are interdependent, but no convenient canonical representation exists that provides a link. This is in contrast to temporal reasoning, in which concepts such as periods and durations are defined in terms of time points.

Context embedding. Spatial reasoning problems are not likely to occur in pure form in practice. They are embedded into a non-spatial context or contain application-specific side constraints. For example, we consider below evolutions of spatial scenarios over time.

Systems. The usual relation constraint approach is not declarative in a strict sense: knowledge is stated in algorithmic or at best meta constraint form. However, typical current constraint solving systems focus on domain reduction, and rarely provide facilities to easily access and modify the constraint network, which is required for enforcing PC. Custom-built reasoning systems are needed.

QSR with relation variables responds to these points. Aspect integration is facilitated by stating additional inter-aspect constraints. These constraints are dealt with on the same level and by the same constraint propagation algorithms as the intra-aspect constraints. Similarly, a spatial reasoning problem can be

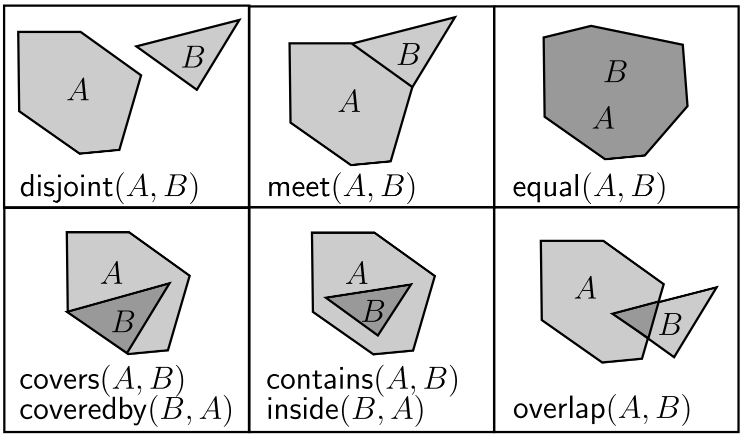

Figure 9.1: RCC-8 relations (2D example)

embedded or extended by viewing it as a set of subconstraints in a larger constraint satisfaction problem. We demonstrate this point by an extensive study of qualitative spatial simulation in Chapter 10.

Finally, the relation variable approach results in a conceptually plain, single-level constraint satisfaction problem. Checking satisfiability corresponds directly to searching a solution. The problem constraints, which are extensionally defined, can be propagated by any chosen method; generalised arc-consistency (GAC, Def. 2.1.5) can be established with membership rules, for example. In fact, any sufficiently expressive constraint solving method can be applied, a search&propagation-based solver is just one option.

9.2 Topological Reasoning with RCC-8

Reasoning about topology is one of the best-known cases of spatial reasoning. It was introduced by [Egenhofer, 1991] and [Randell et al., 1992b]. The latter in particular studied the *Region Connection Calculus* with 8 relations (RCC-8). We use it in the following to illustrate a number of concepts in QSR.

In RCC-8 one distinguishes eight topological relations between two regions. We denote the eight topological relations between two regions that are distinguishes in RCC-8 by disjoint, meet, overlap, equal, covers, coveredby, contains, inside; their meaning is indicated in Fig. 9.1. We call the resulting set RCC8. These relations are *jointly exhaustive* and *pairwise disjoint*: any two spatial regions are in one and exactly one of the RCC-8 relations to each other.

A spatial topological scenario consists of a set of region names and restrictions on the topological relation for pairs of regions. A scenario is fully specified if exactly one topological relation is given for each region pair.

9.2.1 Composition

Considering the triple $R_{a,b}, R_{b,c}, R_{a,c}$ of relations between some regions a, b, c, one finds that not all triples of RCC-8 relations are semantically feasible. The 193 consistent triples are collected in the so-called composition table, see Figure 9.2 (the entry RCC8 abbreviates the full list). For instance, the ⟨inside, meet, disjoint⟩ means that if region a is *inside* region b and b *'touches'* region c then it is consistent with the topological semantics that region a is *disjoint* of c.

The significance of the composition table comes from the fact that compositional consistency entails global consistency. This was proved in [Bennett, 1998]. If, in a fully specified topological scenario, for all triples of regions the relations between them respect the composition table then the scenario is consistent. Global semantic consistency can hence be established by local reasoning.

9.2.2 Converse Relation

In analogy to the composition table, it is helpful to think of a converse relation table consisting of the eight pairs $\langle R, R^I \rangle$ of RCC-8 relations such that R^I is the converse of R. It is printed in Fig. 9.3.

If equal is agreed upon as the relation of a region with itself then the converse relation table follows from the composition table.

9.3 Modelling QSR with Constraints

We examine now how spatial scenarios can be modelled as constraint satisfaction problems. We continue using RCC-8 as an example, but most of the concepts below are immediately transferable to other spatial aspects.

The set of region names in a topological scenario is from now on denoted by *Regions*.

9.3.1 Relations as Constraints

In this conventional approach, *Regions* is considered to be a set of region variables. Their domain is the set of all spatial regions in the underlying topological space. For example, if we model 2D space then a region variable represents a set of points in the plane; so the domain may be infinite. Information about the topological relation between two regions is expressed as a binary constraint *Rel* that corresponds to a subset of RCC8. One usually writes this in infix notation as

$$\text{constraint} \quad x \; Rel \; y \qquad \text{where} \quad Rel \subseteq \mathsf{RCC8} \quad \text{and} \quad x, y \in Regions.$$

	disjoint	meet	equal	inside	coveredby	contains	covers	overlap
disjoint	RCC8	disjoint meet inside coveredby overlap	disjoint	disjoint meet inside coveredby overlap	disjoint meet inside coveredby overlap	disjoint	disjoint	disjoint meet inside coveredby overlap
meet	disjoint meet contains covers overlap	disjoint meet equal coveredby covers overlap	meet	inside coveredby overlap	meet inside	disjoint	disjoint meet	disjoint meet inside coveredby overlap
equal	disjoint	meet	equal	inside	coveredby	contains	covers	overlap
inside	disjoint	disjoint	inside	inside	inside	RCC8	disjoint meet inside coveredby overlap	disjoint meet inside coveredby overlap
coveredby	disjoint	disjoint meet	coveredby	inside	inside coveredby	disjoint meet contains covers overlap	disjoint meet equal coveredby covers overlap	disjoint meet overlap coveredby overlap
contains	disjoint meet contains covers overlap	contains covers overlap	contains overlap	equal inside coveredby contains covers	contains covers overlap	contains	contains	contains covers overlap
covers	disjoint meet contains covers overlap	meet contains covers overlap	covers	inside coveredby overlap	equal coveredby covers overlap	contains	contains covers	contains covers overlap
overlap	disjoint meet contains covers overlap	disjoint meet contains covers overlap	overlap	inside coveredby overlap	inside coveredby overlap	disjoint meet contains covers overlap	disjoint meet contains covers overlap	RCC8

Figure 9.2: The composition table for the RCC-8 relations

disjoint	disjoint
meet	meet
equal	equal
covers	coveredby
coveredby	covers
contains	inside
inside	contains
overlap	overlap

Figure 9.3: The converse relations for the RCC-8 relations

Such a CSP describes a possibly partially specified scenario. Whether a corresponding fully specified and satisfiable scenario exists is checked by path-consistency and search over the relations. A PC-establishing algorithm revises the constraints between regions according to the converse relation and composition tables of RCC-8, and search branches over disjunctive constraints.

When establishing satisfiability of a scenario, only the constraints are processed, for compositional consistency. The variables remain unassigned.

9.3.2 Relations as Variables

Here we interpret every element of *Regions* as a constant. The topological relation between two regions is a variable with a subset of $\mathsf{RCC8}$ as its domain. Such a relation variable exists for each ordered pair of regions, and we collect all these variables in an array *Rel*. We write an individual relation as

$$\text{variable} \quad Rel[a,b] \qquad \text{where} \quad Rel[a,b] \subseteq \mathsf{RCC8} \quad \text{and} \quad a,b \in Regions.$$

Integrity Constraints

Relation converse and composition in this setting are captured at the constraint level. The binary constraint conv represents the converse relation table:

$$\mathsf{conv}(\,Rel[a,b],\ Rel[b,a]\,) \qquad \text{for all} \quad \{a,b\} \subseteq Regions.$$

The composition table is represented by the ternary constraint comp, with

$$\mathsf{comp}(\,Rel[a,b],\ Rel[b,c],\ Rel[a,c]\,) \qquad \text{for all} \quad \{a,b,c\} \subseteq Regions.$$

In presence of

$$Rel[a,a] = \mathsf{equal} \qquad \text{for all} \quad a \in Regions$$

and a conv constraint for all pairs of different regions, one comp constraint per three different regions suffices.

9.3.3 Discussion

By modelling the items of interest as variables and static information as constraints, the relation variable approach yields plain finite-domain CSPs in which the solutions (i. e., assignments) are relevant. There is a straightforward correspondence between a solution and a fully specified, consistent scenario. Obtaining the latter from a partially specified scenario amounts to the standard task of solving a finite-domain CSP.

Constructing a relation variable model means finding integrity constraints that embody the intended semantics. Once that has been accomplished, the origin or meaning of the constraints plays no role. It is irrelevant for a constraint solver whether comp represents the composition operation in a relation algebra. We discuss examples below in which other restrictions on the relations must be satisfied. There is thus a clear distinction between specification and execution. The relation variable approach is declarative in a strict sense.

Constraint Propagation

The relation variable approach is independent of the particular constraint solving method. We can, however, choose a solver based on search and propagation, and furthermore we could choose a GAC-establishing propagation algorithm.

Path consistency in the relation constraint approach and generalised arc-consistency in the relation variable approach simulate each other. This can be seen by analysing, in both approaches, the removal of one topological relation from the disjunctive constraint $a\ \mathit{Rel}\ b$, or from the domain of the variable $\mathit{Rel}[a, b]$, respectively. The reason in both cases must be the lack of supporting relations between a, c and b, c, for some third region c; that is, compositional consistency.

Time Complexity

It is perhaps not surprising but useful to mention that establishing the respective local consistency in either approach, i. e., PC for the relation constraint approach and GAC for the relation variable approach, requires the same computational effort.

Enforcing PC by an algorithm as the one given in [Mackworth, 1977] requires time in $O(n^3)$ [Mackworth and Freuder, 1985], where n is the number of regions. It is assumed there that one PC step, restricting $a\ \mathit{Rel}\ c$ by $a\ \mathit{Rel}\ b$ and $b\ \mathit{Rel}\ c$, takes constant time.

Analogue reasoning entails that GAC can be enforced in constant time on a single $\mathsf{comp}(\mathit{Rel}[a, b], \mathit{Rel}[b, c], \mathit{Rel}[a, c])$ constraint — observe that the three variables have domains of size at most eight. In this way, the overall time complexity depends only on the number $\binom{n}{3}$ of such constraints; and it is thus in $O(n^3)$ as well.

Related Work

In [Tsang, 1987] the relation variable approach is described for qualitative temporal reasoning, a field similar to QSR. The idea appears not to have caught on, however. One reason is probably that integration in temporal reasoning is simpler because the canonical representation of time points on the real line exists. By referring to its end points, a time interval can directly be related to its duration or another time interval. Space, in contrast, has no such convenient canonical representation — but many aspects to be integrated.

In QSR, the possibility of the relation variable approach is mentioned occasionally in passing, but without examining its potential. For actually modelling and solving QSR problems using relation variables I am only aware of [Apt, 2003, pages 30-33], which deals with a single aspect only.

From the perspective of constraint logic programming, relation variables are discussed in [Lamma et al., 1999]. It is studied there how binary relational reasoning can be embedded into CLP(FD), and how relational (qualitative) information can be combined with quantitative information by constraints.

Finally, it is important to point out that the elaborate heuristics developed for qualitative spatial reasoning, such as those reported in [Renz and Nebel, 2001] for RCC-8 and in [Ligozat, 1998] for the cardinal directions, can be easily integrated into a relation variable constraint solver. In this view, these heuristics are customary variable and value ordering heuristics by type.

9.4 Relation Variables in Use

An essential advantage of the relation variable approach is that the relevant information is available in variables. This means that linking pieces of information reduces to merely stating additional constraints on the variables. In that way, embedding a QSR problem into an application context or adding side restrictions, for example, can be dealt with easily and declaratively.

We now illustrate the use of relation variables. We consider the issue of composite models with several cases of aspect integration. Furthermore, we discuss cases in which a qualitative relation is not atomic but is best represented as a set of atoms. This view leads to the use of *set variables* and constraints, which are well established in constraint programming. Finally, we show how one can elegantly constrain *objects*, not only qualitative relations, with relation variables and array constraints of Chapter 8.

9.4.1 Combining Topology and Size

Following [Gerevini and Renz, 2002], we study scenarios combining topological and size information. We collect information about both these aspects and their link in one CSP.

From now on, n denotes the number of regions.

Topological Aspect

As in Section 9.3.2, the

$n \times n$ array *TopoRel*

of RCC-8 relation variables stores the topological relation between two regions. The integrity constraints $\mathsf{conv}_{\mathrm{RCC8}}$, $\mathsf{comp}_{\mathrm{RCC8}}$ need to hold.

Size Aspect

Relative size of regions is captured by one of relations $\{<, =, >\}$, as in [Gerevini and Renz, 2002]. The

$n \times n$ array *SizeRel*

of variables stores the relative sizes of region pairs. The converse relation and composition tables are straightforward; the integrity constraints are

$$\begin{aligned} \mathsf{conv}_{\mathrm{Size}} &= \{\, (<, >),\ (=, =),\ (>, <)\, \}, \qquad \text{and} \\ \mathsf{comp}_{\mathrm{Size}} &= \{\, (<, <, <),\ (<, =, <),\ \ldots\} \qquad \text{(13 triples).} \end{aligned}$$

Linking the Aspects

The topological relation between two regions is dependent on their relative size. A table with this information is given in [Gerevini and Renz, 2002], it contains rules such as the following:

$$\begin{array}{lll} \mathit{TopoRel}[a, b] = \mathsf{inside} & \text{implies} & \mathit{SizeRel}[a, b] = (<), \\ \mathit{SizeRel}[a, b] = (=) & \text{implies} & \mathit{TopoRel}[a, b] \in \{\mathsf{disjoint}, \mathsf{meet}, \mathsf{overlap}, \mathsf{equal}\}. \end{array}$$

In [Gerevini and Renz, 2002], these rules represent a *meta constraint*; they are used within an algorithm that modifies constraints. Here, we infer instead a proper constraint, linking both aspects. We define $\mathsf{link}_{\mathrm{Topo\&Size}}$ by the set

$$\{\ (\mathsf{inside},\ <),\ (\mathsf{disjoint},\ =),\ \ldots\ \} \qquad \text{(14 pairs)}$$

and state it as

$$\mathsf{link}_{\mathrm{Topo\&Size}}(\ \mathit{TopoRel}[a, b],\ \mathit{SizeRel}[a, b]\,)$$

for all regions a, b.

9.4.1. Example. Let us pick up the combined scenario from [Gerevini and Renz, 2002, p. 14]. Five regions, numbered $0, \ldots, 4$, are constrained by

$$\begin{array}{ll}
\mathit{TopoRel}[0,2] \in \{\mathsf{coveredby}, \mathsf{equal}\}, & \\
\mathit{TopoRel}[1,0] \in \{\mathsf{coveredby}, \mathsf{equal}, \mathsf{overlap}\}, & \mathit{SizeRel}[0,2] \in \{<\}, \\
\mathit{TopoRel}[1,2] \in \{\mathsf{coveredby}, \mathsf{equal}\}, & \mathit{SizeRel}[3,1] \in \{<, =\}, \\
\mathit{TopoRel}[4,3] \in \{\mathsf{coveredby}, \mathsf{equal}\}, & \mathit{SizeRel}[2,4] \in \{<, =\}.
\end{array}$$

Independently, the topological and the size scenarios are consistent while the combined scenario is not. It is pointed out in [Gerevini and Renz, 2002] that naive propagation scheduling schemes (first one aspect, then the other; etc.) do not suffice to detect inconsistency.

A formulation of this scenario as a combined topological & size CSP in the relation variable approach is straightforward. The resulting CSP can be solved by a typical constraint programming platform that supports user-defined constraint propagation algorithms, such as the $\mathrm{ECL}^i\mathrm{PS}^e$ system [Wallace et al., 1997]. $\mathrm{ECL}^i\mathrm{PS}^e$ is focused on search and domain-reducing propagation; and it also provides generic GAC-enforcing propagation algorithm for user-defined constraints.

Alternatively, we can use membership rules to enforce GAC. We can compute such rule sets for $\mathsf{conv}_{\mathrm{Size}}$ and $\mathsf{comp}_{\mathrm{Size}}$ from their respective definitions, by the method described in Section 5.4.4 followed by removing redundant rules (Section 4.3.1). We obtain for example the membership rules

$$\begin{array}{rcl}
\mathsf{comp}_{\mathrm{Size}}(x,y,z),\ x \in \{<,=\},\ y \in \{<,=\} & \rightarrow & z \neq (>), \\
\mathsf{comp}_{\mathrm{Size}}(x,y,z),\ x \in \{<\},\ y \in \{<,=\} & \rightarrow & z \neq (=).
\end{array}$$

The definition of the linking constraint $\mathsf{link}_{\mathrm{Topo\&Size}}$ is already provided in terms of rules by [Gerevini and Renz, 2002]. The rules are not membership rules but each is equivalent to one. For example, the rule "$\mathit{TopoRel}[x,y] = \mathsf{inside}$ implies $\mathit{SizeRel}[x,y] = (<)$" mentioned earlier is equivalent to

$$\mathsf{link}_{\mathrm{Topo\&Size}}(x,y),\ x = \mathsf{inside} \quad \rightarrow \quad y \neq (>),\ y \neq (=). \tag{r}$$

While these rules fully define the $\mathsf{link}_{\mathrm{Topo\&Size}}$ constraint, they do not propagate it sufficiently to establish GAC. From these rules we can, however, compute stronger rules that do enforce GAC, as described in Section 5.3.3. Doing so, we obtain, for example, the membership rules

$$\begin{array}{rcl}
\mathsf{link}_{\mathrm{Topo\&Size}}(x,y),\ x \in \{\mathsf{equal}, \mathsf{coveredby}, \mathsf{inside}\} & \rightarrow & y \neq (>), \\
\mathsf{link}_{\mathrm{Topo\&Size}}(x,y),\ x \in \{\mathsf{contains}, \mathsf{covers}, \mathsf{coveredby}, \mathsf{inside}\} & \rightarrow & y \neq (=)
\end{array}$$

to which rule (r) contributed.

Given the above CSP and a constraint propagation method to enforce GAC on the five involved types of constraints, we verified with the help of an ECL^{i}PS^e implementation that solely executing GAC-propagation for all constraints (i. e., no search) yields failure, which proves that this CSP is inconsistent. □

Aspect Integration with Relation Constraints

For the purpose of aspect integration but within the relation constraint approach, in [Gerevini and Renz, 2002] a new algorithm called Bipath-consistency is proposed. Its principle is the computation of path-consistency for both types of relations in an interleaved fashion while taking into account the interdependency. The $\mathsf{link}_{\text{Topo\&Size}}$ constraint is in essence treated as a *meta constraint* on the algorithm level. Moreover, the Bipath-consistency algorithm fixes in part the order of propagation.

The relation variable method, on the other hand, is declarative; all information is in the five types of constraints. They are handled by repeated, interleaved calls to the same GAC-enforcing algorithm. The actual propagation order is irrelevant for the result.

Bipath-consistency is restricted to combining two types of relations (e. g., two aspects of space). In contrast, the relation variable approach is compositional in the sense that adding a third aspect, such as morphology [Cristani, 1999] or orientation, is straightforward. It amounts to formulating integrity constraints such as conv and comp, linking constraints to each of the already present aspects, and a constraint linking all three aspects. Some of these constraints may be logically redundant.

9.4.2 Combining Cardinal Directions and Topology

Reasoning about orientation, another important aspect of space, is the study of the relation of two objects, the primary and the reference object, with respect to a frame of reference. Orientation requires thus inherently a ternary relation. However, by agreeing on the frame of reference, a binary relation is obtained.

Absolute Orientation

The binary relation approach is realised in the cardinal direction model [Frank, 1992], based on the geographic (compass) directions. Points as well as regions have been studied as the objects to be oriented. The point-based models can be cast in the relation variable approach analogously to topology, Section 9.3.2. For instance, Frank [Frank, 1992] distinguishes for pairs of points the jointly exhaustive and pairwise disjoint relations $\mathsf{N}, \mathsf{NW}, \mathsf{W}, \ldots$, denoting North, Northwest, West, etc. Ligozat [Ligozat, 1998] gives a composition table.

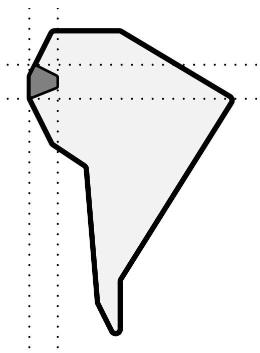

Figure 9.4: Locating South America from within Ecuador

Orienting Regions

In [Goyal and Egenhofer, 1997] and [Skiadopoulos and Koubarakis, 2001] a more expressive model is studied, The oriented objects in this model are regions. The exact shape of the primary region is taken into account, and a ninth atomic relation B exists, describing overlap of the primary region and the axes-parallel minimum bounding box of the reference region. *Sets* of the atomic relations are then used to describe directional information.

In this way, for example, the position of South America for an observer located in Ecuador can be fully described by the (set of) directions 'here', north, northeast, east, southeast, south; see Fig. 9.4.

Relation variables for such directional information of regions are thus naturally *set variables*. They take their value from a set of sets of constants, unlike the relation variables for topology and size whose domain is a set of atomic constants.

For each pair a, b of regions, the direction is a relation variable

$$DirRel[a, b] \in \mathcal{P}(\mathsf{Dir}) \qquad \text{with} \quad \mathsf{Dir} = \{\mathsf{B}, \mathsf{N}, \mathsf{NW}, \ldots, \mathsf{NE}\}$$

(to be read as "region a seen from region b is in ..."). $\mathcal{P}$ denotes the power set function. So we have for example

$$DirRel[\mathsf{south\_america}, \mathsf{ecuador}] = \{\mathsf{B}, \mathsf{N}, \mathsf{NE}, \mathsf{E}, \mathsf{SE}, \mathsf{S}\}, \quad \text{and}$$
$$DirRel[\mathsf{ecuador}, \mathsf{south\_america}] = \{\mathsf{B}\}.$$

Integrity Constraints

A specific restriction on the set values that $DirRel[a, b]$ can take, arises if a, b are internally connected regions, which is often assumed. Only 218 of the 512 subsets of Dir are then semantically possible. This knowledge can be represented in a *unary integrity constraint*

$$\mathsf{connected}(\, Rel[a, b]\,) \qquad \text{for all} \quad \{a, b\} \subseteq Regions.$$

For example, connected allows $\{\mathsf{N}, \mathsf{NW}, \mathsf{W}\}$ but excludes $\{\mathsf{N}, \mathsf{S}\}$.

The constraints comp and conv can be derived from studies of composition [Skiadopoulos and Koubarakis, 2001] and converse [Cicerone and Felice, 2004] (it is outside of our focus whether these are the only integrity constraints needed).

Integration with Topology

Let us consider linking directional information to topology. The relevant knowledge could be expressed by rules such as

$$\begin{array}{lll} \mathit{TopoRel}[a,b] \in \{\mathsf{equal}, \mathsf{inside}, \mathsf{coveredby}\} & \text{implies} & \mathit{DirRel}[a,b] = \{\mathsf{B}\}, \\ \mathit{TopoRel}[a,b] \in \{\mathsf{contains}, \mathsf{covers}\} & \text{implies} & \mathit{DirRel}[a,b] \supseteq \{\mathsf{B}\}. \end{array}$$

These rules define a linking constraint $\mathsf{link}_{\text{Topo\&Dir}}$, to be stated as

$$\mathsf{link}_{\text{Topo\&Dir}}(\ \mathit{TopoRel}[a,b],\ \mathit{DirRel}[a,b]\ )$$

for all regions a, b. We now have some components of a combined cardinal directions & topology model. It can be given to any sufficiently expressive constraint solver, which in particular would provide constraints on set variables.

Constraint solving with set variables is an established subject in research on constraint programming; it is discussed in [Gervet, 1997], for example. Many contemporary constraint programming systems support set variables.

9.4.3 Cyclic Ordering of Orientations

From the several formalisations of orientation information with an explicit frame of reference, let us examine the approach to cyclic ordering of 2D orientations of [Isli and Cohn, 2000]. Here, the spatial objects are orientations, i. e. directed lines. At the root of the framework is the qualitative classification of the angle $\sphericalangle(a, b)$ between the two orientations a and b by

$$\mathsf{Or}(\sphericalangle(a,b)) = \begin{cases} \mathsf{e} \quad \text{(equal)} & \text{if} \quad \alpha = 0, \\ \mathsf{l} \quad \text{(left)} & \text{if} \quad 0 < \alpha < \pi, \\ \mathsf{o} \quad \text{(opposite)} & \text{if} \quad \alpha = \pi, \\ \mathsf{r} \quad \text{(right)} & \text{if} \quad \pi < \alpha < 2\pi \end{cases}$$

into the jointly exhaustive and pairwise disjoint relations $\mathsf{e}, \mathsf{l}, \mathsf{o}, \mathsf{r}$; see Fig. 9.5 for an illustration. For three orientations a, b, c, we now consider the triple

$$\langle\ \mathsf{Or}(\sphericalangle(b,a)),\ \mathsf{Or}(\sphericalangle(c,b)),\ \mathsf{Or}(\sphericalangle(c,a))\ \rangle.$$

Of all 4^3 triples over $\{\mathsf{e}, \mathsf{l}, \mathsf{o}, \mathsf{r}\}$, only 24 combinations are geometrically possible; we denote this set by Cyc; Fig. 9.6 shows three of its elements.

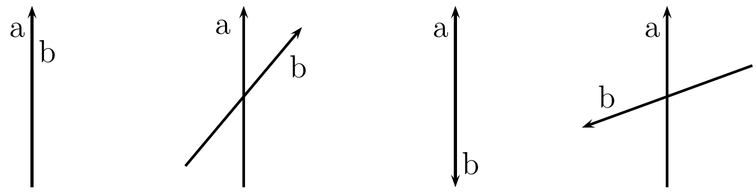

Figure 9.5: The relations $\mathsf{e}, \mathsf{l}, \mathsf{o}, \mathsf{r}$ of a pair of orientations

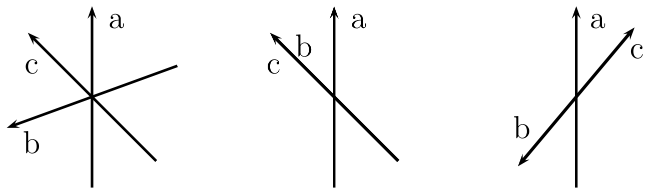

Figure 9.6: The Cyc relations lrl, lel, and rol of a triple of orientations

Such cyclic ordering information can be expressed within the relation variable approach in an array *CycRel* that in particular is ternary. We have thus a relation variable

$$CycRel[a, b, c] \in \mathsf{Cyc} \qquad \text{with} \quad \mathsf{Cyc} = \{\mathsf{lrl}, \mathsf{orl}, \ldots, \mathsf{rle}\}$$

for every three orientations a, b, c. The integrity constraints here are

$$\mathsf{conv}(\ CycRel[a, b, c],\ CycRel[a, c, b]\ ),$$

$$\mathsf{comp}(\ CycRel[a, b, c],\ CycRel[a, c, d],\ CycRel[a, b, d]\ ),$$

and a new constraint

$$\mathsf{rotate}(\ CycRel[a, b, c],\ CycRel[c, a, b]\ ).$$

Details and definitions can be found in [Isli and Cohn, 2000].

Working within the relation constraint approach, Isli and Cohn construct a new algorithm called *s4c* that enforces 4-consistency [Freuder, 1978] on the ternary relation constraints that correspond to *CycRel*. They are able to prove that this algorithm decides consistency, i. e., 2D geometric feasibility, of fully specified scenarios. The *s4c* algorithm uses exactly the information that we represent in the conv, comp and $\mathsf{rotation}$ constraints. Consequently, we can conclude that in our relation variable model these constraints guarantee geometric consistency.

We hypothesise further that *s4c* in the relation constraint model propagates at most as much information as a GAC-enforcing algorithm does in our relation variable model. Intuitively, this should be clear: every possible reduction of a disjunctive constraint in the relation constraint model corresponds to a domain reduction of a relation variable in our model.

9.4.4 Combining Cardinal Direction with Relative Orientation

The exchange of information between a cardinal direction model for pairs of points as mentioned in Section 9.4.2 and a relative orientation model for triples of points derived from the formalisation in [Freksa and Zimmermann, 1992] is studied in [Isli, 2003, Isli, 2004]. A rough example for the kind of reasoning that should be facilitated is the following. Suppose the points a, b, c in the plane are such that

- c is to the left of the directed line ab, and
- b is north of a,

then c is south-west, west, or north-west of a and b

This integration task is quite analogous to the case of combining topology and size (Section 9.4.1), and so is the solution. Isli, working with relation constraints, proposes a new algorithm for this integration issue.

We formulate a relation variable model. The cardinal direction subproblem can straightforwardly be expressed in this approach; we omit the obvious details here. The relative orientation subproblem leads to a model similar to that of orientations in the preceding section; in particular, it is based on a ternary array. The arrays in the combined model are

$$\begin{array}{ll} \mathit{CDirRel}, & \text{which is an } n \times n \text{ array, and} \\ \mathit{ROrientRel}, & \text{which is an } n \times n \times n \text{ array,} \end{array}$$

if we assume n points.

For linking the two models, independent functions for both directions of the information transfer are described in [Isli, 2004]. They can be transformed into the two constraints

$$\mathsf{link}_{\mathrm{CD}\rightarrow\mathrm{RO}}(\, \mathit{CDirRel}[a,b],\ \mathit{CDirRel}[b,c],\ \mathit{ROrientRel}[a,b,c]\,),$$
$$\mathsf{link}_{\mathrm{CD}\leftarrow\mathrm{RO}}(\, \mathit{ROrientRel}[a,b,c],\ \mathit{CDirRel}[a,b],\ \mathit{CDirRel}[b,c],\ \mathit{CDirRel}[a,c]\,)$$

on triples of points a, b, c.

In the relation constraint approach it is necessary to treat the information in $\mathsf{link}_{\mathrm{CD}\rightarrow\mathrm{RO}}$ and $\mathsf{link}_{\mathrm{CD}\leftarrow\mathrm{RO}}$ as meta constraints. An embedding algorithm is given in [Isli, 2004]. This algorithm moreover integrates a variant of the *s4c* algorithm of [Isli and Cohn, 2000] and a path-consistency algorithm.

Using relation variables, it suffices to state the constraints and provide a generic GAC-enforcing algorithm. Also, by taking into account the semantics, for a given triple of points, the first constraint $\mathsf{link}_{\mathrm{CD}\rightarrow\mathrm{RO}}$ should just be the restriction of the second constraint $\mathsf{link}_{\mathrm{CD}\leftarrow\mathrm{RO}}$ in which the variable $\mathit{CDirRel}[a,c]$ is projected away. The former constraint is then redundant, and we end up with one linking constraint

$$\mathsf{link}_{\mathrm{CD\&RO}}(\, \mathit{ROrientRel}[a,b,c],\ \mathit{CDirRel}[a,b],\ \mathit{CDirRel}[b,c],\ \mathit{CDirRel}[a,c]\,).$$

On the grounds that both the relation variable and the relation constraint approach are based on the same semantic information, for one embedded in an algorithm, for the other in constraints, we conclude that both accept exactly the same point configuration scenarios.

9.4.5 Object Variables and Array Constraints

In the relation variable model, spatial objects are denoted by constants. An *object variable*, whose domain is the set of object constants, has thus a different meaning here than in the relation constraint approach. This issue is best demonstrated by an example. Suppose we wish to identify two regions among all given regions such that

- the first is smaller than the second, and
- they are disconnected or externally connected.

We use topological and size information as formalised as in Section 9.4.1, so we have arrays *SizeRel* and *TopoRel* recording the qualitative relations. Let *Regions* be the set of the n region constants. We define

region variables x_1, x_2

whose domain is the set *Regions*. They are constrained by

$$\begin{aligned} &\mathit{SizeRel}[x_1, x_2] = (<), \\ &\mathit{TopoRel}[x_1, x_2] \in \{\mathsf{DC}, \mathsf{EC}\}. \end{aligned} \qquad (C)$$

C is a constraint on the variables x_1, x_2 and on all size relation variables in the array *SizeRel*. Namely, region constants $r_1, r_2 \in \mathit{Regions}$ must be assigned to x_1, x_2 such that the selected relation variable $\mathit{SizeRel}[r_1, r_2]$ is assigned a '$<$'. This constraint is an array constraint, which we study in Chapter 8.

9.5 Implementation

To examine the feasibility of the relation variable approach, we implemented it in ECL^{i}PS^e. Specifically, we deal with topology, size, their integration, and the cardinal directions. Constraint propagation of all constraints is realised by membership rules that establish generalised arc-consistency.

The implementation also incorporates the specialised heuristics for reasoning with RCC-8 [Ligozat, 1998] and the cardinal directions [Renz and Nebel, 2001]. Note that these techniques are heuristics, not decision procedures, in our context. With relation variables, they fall into the customary class of variable and value ordering heuristics in constraint programming search. Our experience is that the use of these heuristics massively reduces the solving time.

The implementation is used in Example 9.4.1; it is also the basis of the implementation of *dynamic* spatial reasoning which we discuss in Chapter 10.

9.6 Final Remarks

We discussed here an alternative formulation of qualitative spatial reasoning problems as constraint satisfaction problems. Contrary to the conventional approach, we model qualitative relations as variables. Uncertain relational information is naturally expressed by variables with domains; consistency of this information is naturally expressed by static constraints. The propagation of these constraints is a well-understood issue in constraint programming; corresponding generic algorithms are provided by many constraint solving systems; also membership rules can be used to that end.

While the principle of the relation variable approach has been described earlier, for instance for modelling qualitative temporal reasoning, the advantages of applying it to QSR, especially for integration tasks, have so far very rarely been realised. We argued that several algorithms that are custom-designed for integrating spatial aspects become redundant if a relation variable model and a generic GAC-establishing constraint propagation algorithm is used: the Bipath-consistency algorithm of [Gerevini and Renz, 2002], the *s4c* algorithm of [Isli and Cohn, 2000], the algorithm combining *s4c* and a path-consistency algorithm of [Isli, 2004].

We showed how the relation variable approach can accommodate composite qualitative relations as investigated in [Cicerone and Felice, 2004, Skiadopoulos and Koubarakis, 2001] with the help of set variables and constraints. Extending or combining a relation variable model often consists only in defining appropriate constraints, contrary to what is the case in the relation constraint approach where new algorithms must be designed.

Finally, let us remark that the strictly declarative model obtained within the relation variable approach can be solved by any sufficiently expressive CSP solver. This includes typical CP systems based on search and propagation, but for instance also solvers based on local search.

Chapter 10

Qualitative Simulation

10.1 Introduction

Qualitative simulation deals with the reasoning about possible evolutions in time of the models capturing qualitative information. One assumes that time is discrete and that at each stage only changes adhering to some desired format can occur. [Kuipers, 2001] discusses qualitative simulation in the first framework, while *qualitative spatial simulation* is considered in [Cui et al., 1992].

The aim of this chapter is to show how qualitative simulation in the second approach to qualitative reasoning (exemplified by qualitative temporal and spatial reasoning) can be naturally captured by means of temporal logic and constraint satisfaction problems modelled according to the relation variable approach. The resulting framework allows us to describe various complex forms of behaviour, for example a simulation of a throw of a ball into a box, a simulation of the movements of a discus thrower, or a solution to a piano movers problem. The relevant constraints are formulated using a variant of linear temporal logic with both past and future temporal operators. Once such temporal formulas are translated into the customary constraints, standard techniques of constraint programming can be used to generate the appropriate simulations and to answer various queries about them. To support this claim, we implemented this approach in the ECL^{i}PS^e programming system and discuss here experiments.

10.2 Simulation Constraints

10.2.1 Intra-state Constraints

To describe formally qualitative simulations, we define first intra-state and inter-state constraints. A qualitative simulation is then a CSP that consists of 'stages' that all satisfy the intra-state constraints. Moreover, this CSP satisfies the inter-state constraints that link variables appearing in various stages.

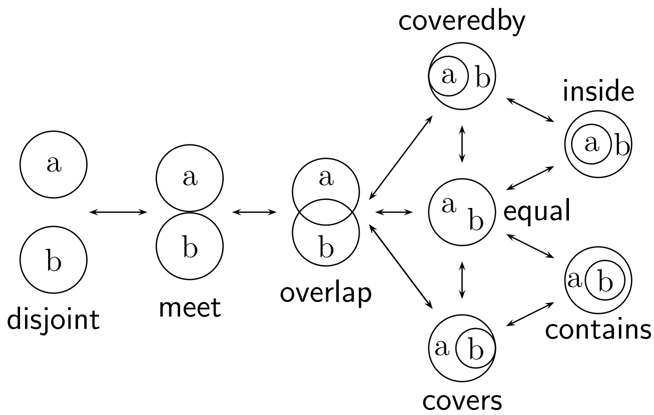

Figure 10.1: The neighbourhood relation for the RCC-8 relations

For presentational reasons, we restrict ourselves from now on to simple *binary* qualitative relations (e. g., topology, size). This is no fundamental limitation; the principles we outline extend easily to the non-binary case (e. g., the ternary orientation relation).

We assume that we have at our disposal

- a finite *set of qualitative relations* $\mathcal{Q}$, with a special element denoting the relation of an object to itself,
- the integrity constraints in a relation variable model, such as a ternary *composition* relation comp and a binary *converse* relation conv,
- a *neighbourhood* relation $\mathsf{neighbour}$ between the elements of $\mathcal{Q}$ that describes which 'atomic' changes in the qualitative relations are admissible.

10.2.1. Example. Take the qualitative spatial reasoning with topology introduced in [Egenhofer, 1991] and [Cui et al., 1992], and discussed in Section 9.2. The set of qualitative relations is the set $\mathsf{RCC8}$, i. e.,

$$\mathcal{Q} = \{\mathsf{disjoint}, \mathsf{meet}, \mathsf{equal}, \mathsf{covers}, \mathsf{coveredby}, \mathsf{contains}, \mathsf{inside}, \mathsf{overlap}\}.$$

The composition and converse relations are given in Figures 9.2 and 9.3.

The neighbourhood relation is depicted in Figure 10.1. We assume here that during the simulation the objects can change their size. If we wish to disallow this possibility, then the pairs $(\mathsf{equal}, \mathsf{coveredby})$, $(\mathsf{equal}, \mathsf{covers})$, $(\mathsf{equal}, \mathsf{inside})$, $(\mathsf{equal}, \mathsf{contains})$ and their converses should be excluded from the above neighbourhood relation. $\square$

We fix now a sequence $\mathcal{O}$ of objects of interest. By a *qualitative array* we mean a two-dimensional array Q on $\mathcal{O} \times \mathcal{O}$ such that

- for each pair of objects $\mathtt{A}, \mathtt{B} \in \mathcal{O}$, $Q[\mathtt{A}, \mathtt{B}]$ is a variable with the domain included in $\mathcal{Q}$,
- the integrity constraints hold on Q, so for each triple of objects $\mathtt{A}, \mathtt{B}, \mathtt{C}$ the following *intra-state constraints* are satisfied:
  - *reflexivity*: $Q[\mathtt{A}, \mathtt{A}] = \mathsf{equal}$,
  - *transposition*: $\mathsf{conv}(Q[\mathtt{A}, \mathtt{B}], Q[\mathtt{B}, \mathtt{A}])$,
  - *composition*: $\mathsf{comp}(Q[\mathtt{A}, \mathtt{B}], Q[\mathtt{B}, \mathtt{C}], Q[\mathtt{A}, \mathtt{C}])$.

Each qualitative array determines a unique CSP. Its variables are $Q[\mathtt{A}, \mathtt{B}]$, with $\mathtt{A}$ and $\mathtt{B}$ ranging over the sequence of the assumed objects $\mathcal{O}$. The domains of these variables are appropriate subsets of $\mathcal{Q}$. In what follows we represent each stage t of a simulation by a CSP $\mathcal{P}_t$ uniquely determined by a qualitative array Q_t. Here t is a variable ranging over the set of natural numbers that represents discrete time. Instead of $Q_t[\mathtt{A}, \mathtt{B}]$ we write $Q[\mathtt{A}, \mathtt{B}, t]$, which reflects that, in fact, we deal with a *ternary* array.

10.2.2 Inter-state Constraints

To describe the inter-state constraints we use as *atomic formulas* statements of the form

$$Q[\mathtt{A}, \mathtt{B}] \ ? \ q,$$

where $? \in \{=, \neq\}$ and $q \in \mathcal{Q}$, and 'true', and employ a *temporal logic* with four temporal operators,

$\bigcirc$ (next time),
$\Diamond$ (eventually),
$\Box$ (from now on), and
U (until),

and their 'past' counterparts, $\bigcirc^{-1}$, $\Diamond^{-1}$, $\Box^{-1}$ and S (since). While it is known that past time operators can be eliminated, their use results in more succinct (and in our case more intuitive) specifications, see, e. g., [Markey et al., 2002].

We use as *inter-state constraints* formulas of the form $\phi \rightarrow \bigcirc\psi$, where ϕ contains only the past time operators and ψ contains only the future time operators. Both ϕ and ψ are built out of atomic formulas using propositional connectives, and temporal operators of the appropriate kind. Intuitively, at each time instance t, each interstate constraint $\phi \rightarrow \bigcirc\psi$ links the 'past' CSP $\bigcup_{i=0}^{t} \mathcal{P}_i$ with the 'future' CSP $\bigcup_{i=t+1}^{t_{\max}} \mathcal{P}_i$, where $t_{\max}$ is the fixed maximum length of the simulation. So we interpret ϕ in the interval $[0..t]$, and ψ in the interval $[t+1 \,..\, t_{\max}]$.

We explain the meaning of a past or future temporal formula ϕ with respect to the underlying spatial array Q in an interval $[s..t]$, for which we stipulate $s \leqslant t$. We write $\models_{[s..t]} \phi$ to express that ϕ holds in the interval.

Propositional connectives. These are defined as expected, in particular independently of the 'past' or 'future' aspect of the formula.

$$
\begin{array}{ll}
\models_{[s..t]} \mathsf{true} & \text{true,} \\
\models_{[s..t]} \neg\phi & \text{if not } \models_{[s..t]} \phi, \\
\models_{[s..t]} \phi_1 \vee \phi_2 & \text{if } \models_{[s..t]} \phi_1 \text{ or } \models_{[s..t]} \phi_2.
\end{array}
$$

Conjunction $\phi_1 \wedge \phi_2$ and implication $\phi_1 \rightarrow \phi_2$ are defined analogously.

Future formulas. Intuitively, we are at the lower bound of the time interval and move only forward in time.

$$
\begin{array}{ll}
\models_{[s..t]} Q[A,B]\,?\,c & \text{if } Q[A,B,s]\,?\,c \text{ where } ? \in \{=, \neq\}, \\
\models_{[s..t]} \bigcirc\phi & \text{if } \models_{[r..t]} \phi \text{ and } r = s+1, r \leqslant t, \\
\models_{[s..t]} \Box\phi & \text{if } \models_{[r..t]} \phi \text{ for all } r \in [s..t], \\
\models_{[s..t]} \Diamond\phi & \text{if } \models_{[r..t]} \phi \text{ for some } r \in [s..t], \\
\models_{[s..t]} \chi \,\mathsf{U}\, \phi & \text{if } \models_{[r..t]} \phi \text{ for some } r \in [s..t] \\
 & \quad \text{and } \models_{[u..t]} \chi \text{ for all } u \in [s \,..\, r-1].
\end{array}
$$

Past formulas. We are here at the upper bound of the time interval and move backward.

$$
\begin{array}{ll}
\models_{[s..t]} Q[A,B]\,?\,c & \text{if } Q[A,B,t]\,?\,c \text{ where } ? \in \{=, \neq\}, \\
\models_{[s..t]} \bigcirc^{-1}\phi & \text{if } \models_{[s..r]} \phi \text{ and } r = t-1, s \leqslant r, \\
\models_{[s..t]} \Box^{-1}\phi & \text{if } \models_{[s..r]} \phi \text{ for all } r \in [s..t], \\
\models_{[s..t]} \Diamond^{-1}\phi & \text{if } \models_{[s..r]} \phi \text{ for some } r \in [s..t], \\
\models_{[s..t]} \chi \,\mathsf{S}\, \phi & \text{if } \models_{[s..r]} \phi \text{ for some } r \in [s..t] \\
 & \quad \text{and } \models_{[u..t]} \chi \text{ for all } u \in [r+1 \,..\, t].
\end{array}
$$

Observe here that the formula $Q[\mathtt{A},\mathtt{B}]\,?\,q$ is interpreted in two ways, depending on whether it is in the 'past' or in the 'future'.

We also use the following abbreviations,

$$Q[\mathtt{A},\mathtt{B}] \in \{q_1, \ldots, q_k\} \quad \text{for} \quad (Q[\mathtt{A},\mathtt{B}] = q_1) \vee \ldots \vee (Q[\mathtt{A},\mathtt{B}] = q_k),$$

and

$$Q[\mathtt{A},\mathtt{B}] \notin \{q_1, \ldots, q_k\} \quad \text{for} \quad (Q[\mathtt{A},\mathtt{B}] \neq q_1) \wedge \ldots \wedge (Q[\mathtt{A},\mathtt{B}] \neq q_k).$$

Furthermore, we use bounded quantification to abbreviate the following cases of disjunctions and conjunctions, i. e.,

$$\forall \mathtt{A} \in \{o_1, \ldots, o_k\}.\ \phi(\mathtt{A}) \quad \text{for} \quad \phi(o_1) \wedge \ldots \wedge \phi(o_k),$$

and

$$\exists \mathtt{A} \in \{o_1, \ldots, o_k\}.\ \phi(\mathtt{A}) \quad \text{for} \quad \phi(o_1) \vee \ldots \vee \phi(o_k).$$

As usual, in $\phi(\mathtt{A})$, $\mathtt{A}$ denotes a placeholder (free variable), and $\phi(o_i)$ is obtained by replacing $\mathtt{A}$ in all its occurrences by o_i.

10.2.3 Examples for Inter-state Constraints

Let us now illustrate the syntax of inter-state constraints by examples. We begin with some 'domain independent' inter-state constraints.

Atomic changes. In each transition only 'atomic' changes can occur. Given an element q of $\mathcal{Q}$, we define

$$\mathsf{neighbour}(q) = \{\, a \mid (q, a) \in \mathsf{neighbour} \,\}.$$

So $\mathsf{neighbour}(q)$ is the set of the qualitative relations that are in the conceptual neighbourhood of relation q. The above inter-state constraint is then formalised as the set of formulas

$$Q[\mathtt{A}, \mathtt{B}] = q \;\rightarrow\; \bigcirc Q[\mathtt{A}, \mathtt{B}] \in \{q\} \cup \mathsf{neighbour}(q),$$

with $\mathtt{A}, \mathtt{B}$ ranging over the sequence $\mathcal{O}$ of considered objects, and q ranging over the set of relations $\mathcal{Q}$.

Non-circularity. No looping happens during the simulation. This is formalised as the set of the following formulas

$$\big(\forall \mathtt{A}, \mathtt{B} \in \mathcal{O}.\ Q[\mathtt{A}, \mathtt{B}] = q(\mathtt{A}, \mathtt{B})\big) \;\rightarrow\; \bigcirc \Box\, \exists \mathtt{A}, \mathtt{B} \in \mathcal{O}.\ Q[\mathtt{A}, \mathtt{B}] \neq q(\mathtt{A}, \mathtt{B}),$$

where q is a mapping of the pairs $\mathtt{A}, \mathtt{B}$ to $\mathcal{Q}$. If we drop $\Box$ here, we formalise the *perpetual change* inter-state constraint stating that in each transition some change takes place.

Next, we provide examples of 'domain dependent' inter-state constraints.

Phagocytosis. (Taken from [Cui et al., 1992].) As soon as an amoeba has absorbed a food particle, the food remains inside the amoeba. This inter-state constraint is formalised as:

$$Q[\mathit{food}, \mathit{amoeba}] = \mathsf{coveredby} \;\rightarrow\; \bigcirc\Box Q[\mathit{food}, \mathit{amoeba}] \neq \mathsf{overlap}.$$

Note that in presence of the intra-state neighbourhood relation depicted in Fig. 10.2 and used when the objects do not change the size, it is sufficient to postulate that

$$Q[\mathit{food}, \mathit{amoeba}] = \mathsf{coveredby} \;\rightarrow\; \bigcirc Q[\mathit{food}, \mathit{amoeba}] \neq \mathsf{overlap}.$$

Indeed, by the form of this neighbourhood relation, if for some t we have $Q[\mathit{food}, \mathit{amoeba}, t] = \mathsf{coveredby}$, then the situation $Q[\mathit{food}, \mathit{amoeba}, t'] = \mathsf{overlap}$ for some $t' > t$ could only happen if $Q[\mathit{food}, \mathit{amoeba}, t' - 1] = \mathsf{coveredby}$.

We consider a model of phagocytosis in detail in Section 10.5.2.

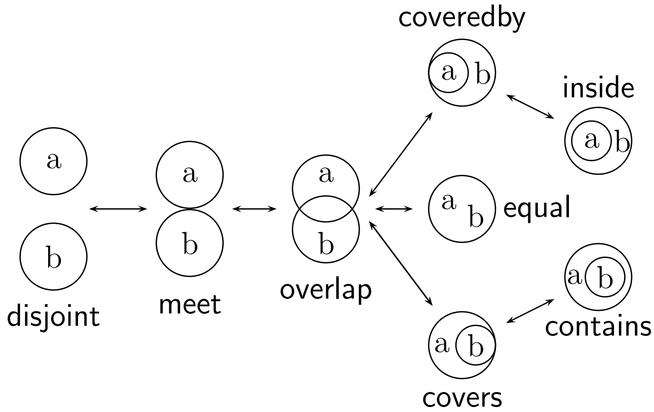

Figure 10.2: An alternative neighbourhood relation for the RCC-8 relations

Ball in a box. Suppose we wish to model that, if some ball is outside some box, it will eventually be inside the box (i. e., inside or $\mathsf{coveredby}$). Afterwards it will remain in the box, though may change its shape. This can be described by the following formulas:

$$Q[\mathit{ball}, \mathit{box}] = \mathsf{disjoint} \;\rightarrow\; \bigcirc\Diamond Q[\mathit{ball}, \mathit{box}] \in \{\mathsf{inside}, \mathsf{coveredby}\},$$

$$Q[\mathit{ball}, \mathit{box}] \in \{\mathsf{inside}, \mathsf{coveredby}\} \;\rightarrow\; \bigcirc\Box Q[\mathit{ball}, \mathit{box}] \in \{\mathsf{inside}, \mathsf{coveredby}, \mathsf{equal}\}.$$

As in the previous example, if we assume that the objects do not change their size, that is, use the neighbourhood relation defined in Figure 10.2, then we can replace the second formula by a simpler one,

$$Q[\mathit{ball}, \mathit{box}] \in \{\mathsf{inside}, \mathsf{coveredby}\} \;\rightarrow\; \bigcirc Q[\mathit{ball}, \mathit{box}] \neq \mathsf{overlap}.$$

Rotations. As soon as an object A starts moving around B, it continues to move in the same direction (either clockwise or counterclockwise). To formalise this constraint, we use qualitative reasoning about the cardinal directions

$$\mathsf{Dir} = \{\mathsf{N}, \mathsf{NE}, \mathsf{E}, \mathsf{SE}, \mathsf{S}, \mathsf{SW}, \mathsf{W}, \mathsf{NW}, \mathsf{EQ}\}$$

with the obvious meaning (EQ is the identity relation). [Ligozat, 1998] provides the composition table for this form of qualitative reasoning. We introduce a relation $\mathit{move}_{\mathrm{CW}}$ (move clockwise):

$$\mathit{move}_{\mathrm{CW}} = \{\; (\mathsf{N}, \mathsf{NE})(\mathsf{NE}, \mathsf{E}), (\mathsf{E}, \mathsf{SE}), (\mathsf{SE}, \mathsf{S}), (\mathsf{S}, \mathsf{SW}), (\mathsf{SW}, \mathsf{W}), (\mathsf{W}, \mathsf{NW}), (\mathsf{NW}, \mathsf{N}) \;\},$$

and use $\mathsf{neighbour}' = \mathit{move}_{\mathrm{CW}} \cup \mathit{move}_{\mathrm{CW}}^{-1}$ as the neighbourhood relation (where $\mathit{move}_{\mathrm{CW}}^{-1}$ describes counterclockwise moves). The above inter-state constraint is now formalised by the set of formulas

$$\phi_{\mathtt{P}} \wedge \bigcirc^{-1}\left(\neg\phi_{\mathtt{P}}\,\mathsf{S}\,(\phi_{\mathtt{Q}} \wedge \bigcirc^{-1}\phi_{\mathtt{P}})\right) \;\rightarrow\; \phi_{\mathtt{P}}\,\mathsf{U}\,\phi_{\mathtt{Q}},$$

where $\phi_{\mathtt{Rel}}$ denotes $Q[A,B] = \mathtt{Rel}$ and $(\mathtt{P},\mathtt{Q})$ ranges over $\mathsf{neighbour}'$.

Navigation. A ship is required to navigate around three buoys along a specified course. The position of the buoys are fixed (Fig. 10.3). We have the permanent invariants

$$\begin{aligned} &Q(\mathit{buoy}_a, \mathit{buoy}_c) = \mathsf{NW},\\ &Q(\mathit{buoy}_a, \mathit{buoy}_b) = \mathsf{SW},\\ &Q(\mathit{buoy}_b, \mathit{buoy}_c) = \mathsf{NW}. \end{aligned}$$

Objects occupy different spaces

$$\forall \mathtt{A},\mathtt{B} \in \mathcal{O}.\ \mathtt{A} \neq \mathtt{B} \rightarrow Q(\mathtt{A},\mathtt{B}) \neq \mathsf{EQ}.$$

The initial position of the ship is south of buoy_a,

$$Q(\mathit{ship}, \mathit{buoy}_a) = \mathsf{S}.$$

The ship is required to follow a path around the buoys. We specify

$$\begin{aligned} &\Diamond(Q[\mathit{ship}, \mathit{buoy}_A] = \mathsf{W} \quad \wedge\\ &\quad \Diamond(Q[\mathit{ship}, \mathit{buoy}_B] = \mathsf{N} \quad \wedge\\ &\quad\quad \Diamond(Q[\mathit{ship}, \mathit{buoy}_C] = \mathsf{E} \quad \wedge\\ &\quad\quad\quad \Diamond(Q[\mathit{ship}, \mathit{buoy}_C] = \mathsf{S} \quad)))), \end{aligned}$$

to hold at the interval $[0 \mathinner{..} t_{\max}]$. A tour of 14 steps exists; in Fig. 10.3, the positions required to be visited are marked with bold circles.

Discus thrower. A discus thrower (T) makes three full rotations before releasing the disc (D) in northern direction. To specify this behaviour we use spatial reasoning combined with the reasoning about the cardinal directions, see Section 9.4.2. We model it here as follows. For each pair of objects $\mathtt{A},\mathtt{B}$ we assume that $Q[\mathtt{A},\mathtt{B}] \subseteq \mathsf{RCC8} \times \mathsf{Dir}$, and adopt the Cartesian products of the corresponding neighbourhood and composition tables. To these intra-state constraints, we add the necessary aspect-linking constraints. Next, given the formulas ϕ and χ we define by induction a sequence of formulas ρ_i as follows:

$$\begin{aligned} \rho_0 &= \phi \;\wedge\; \bigcirc^{-1}\Box^{-1}\chi,\\ \rho_{k+1} &= \phi \;\wedge\; \bigcirc^{-1}(\chi\,\mathsf{S}\,\rho_k). \end{aligned}$$

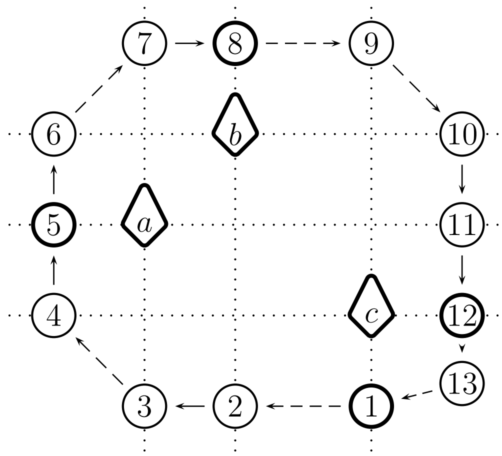

Figure 10.3: Navigation path

Note that when χ implies $\neg\phi$, the formula ρ_k implies that ϕ is true now and has been true precisely at k time instances in the past. So we can formalise the above requirement using the formula

$$\rho_3 \;\rightarrow\; \bigcirc(\phi \mathsf{U} \psi),$$

where

$$\begin{array}{lcl} \phi & \equiv & Q[T,D] = \langle \mathsf{meet}, \mathsf{N} \rangle, \\ \chi & \equiv & Q[T,D] \in \{\, \langle \mathsf{meet}, d \rangle \mid d \in \mathsf{Dir} - \{\mathsf{N}\}\,\}, \quad \text{and} \\ \psi & \equiv & Q[T,D] = \langle \mathsf{disjoint}, \mathsf{N} \rangle. \end{array}$$

10.3 Temporal Formulas as Constraints

We need to explain how a temporal formula is imposed as a constraint on the sequence of CSPs that represent the spatial arrays at consecutive times. We reduce the formula (inter-state constraint) into a conjunction of simple constraints, eliminating the temporal operators in the process.

To be more precise, let us assume a temporal formula $\phi \rightarrow \bigcirc\psi$. Recall that ϕ contains only 'past' time operators and ψ contains only 'future' time operators. Given a CSP $\bigcup_{i=s}^{t} \mathcal{P}_i$, we show how the past temporal logic formula ϕ translates to a constraint $cons^-([s..t], \phi)$, and how a future temporal logic formula ψ translates to a constraint $cons^+([s..t], \psi)$, both on the variables of $\bigcup_{i=s}^{t} \mathcal{P}_i$.

We give first a simple translation based on unfolding, and then an alternative translation that employs array constraints.

10.3.1 Unfolding Translation

To deal with disjunctive formulas in a target constraint language of only conjunctions of constraints, which is the typical case, we assume that the language has Boolean constraints and *reified* versions of some simple comparison and arithmetic constraints. For example, $(x = y) \equiv b$ is a reified equality constraint. b is a Boolean variable reflecting the truth of the constraint $x = y$.

We denote by $cons([s..t], \phi) \equiv b$ the sequence of constraints representing that the formula ϕ has truth value b at interval $[s..t]$. The 'past' or 'future' aspect of a formula is indicated by a marker $^-$ or $^+$, resp., when relevant. The translation of ϕ is initiated by the call $cons([s..t], \phi) \equiv 1$ (where $s \leqslant t$), and we proceed by induction as follows.

Translation for 'future' formulas.

$cons^+([s..t], \mathsf{true}) \equiv b$	is	$b = 1$,
$cons^+([s..t], \neg\phi) \equiv b$	is	$b' = \neg b$, $cons^+([s..t], \phi) \equiv b'$,
$cons^+([s..t], \phi_1 \vee \phi_2) \equiv b$	is	$(b_1 \vee b_2) \equiv b$, $cons^+([s..t], \phi_1) \equiv b_1$, $cons^+([s..t], \phi_2) \equiv b_2$,
$cons^+([s..t], Q[A,B] \mathbin{?} c) \equiv b$	is	$(Q[A,B,s] \mathbin{?} c) \equiv b$ where $? \in \{=, \neq\}$,
$cons^+([s..t], \bigcirc\phi) \equiv b$	is	$(b_1 \wedge b_2) \equiv b$, $(s + 1 \leqslant t) \equiv b_1$, $(s + 1 = r) \equiv b_1$, $cons^+([r..t], \phi) \equiv b_2$,
$cons^+([s..t], \Box\phi) \equiv b$	is	$(\bigwedge_{r \in s..t} b_r) \equiv b$, $cons^+([r..t], \phi) \equiv b_r$ for all $r \in [s..t]$,
$cons^+([s..t], \Diamond\phi) \equiv b$	is	$(\bigvee_{r \in s..t} b_r) \equiv b$, $cons^+([r..t], \phi) \equiv b_r$ for all $r \in [s..t]$,
$cons^+([s..t], \chi \mathbin{\mathsf{U}} \phi) \equiv b$	is	$cons^+([r..t], \phi \vee \chi \wedge \bigcirc(\chi \mathbin{\mathsf{U}} \phi)) \equiv b_r$.

Translation for 'past' formulas. The definition of $cons^-([s..t], \phi) \equiv b$ is entirely symmetric to that of $cons^+([s..t], \phi) \equiv b$ except for the backward perspective. So we have

$cons^-([s..t], Q[A,B] \mathbin{?} c) \equiv b$	is	$(Q[A,B,t] \mathbin{?} c) \equiv b$ where $? \in \{=, \neq\}$,
$cons^-([s..t], \bigcirc^{-1}\phi) \equiv b$	is	$(b_1 \wedge b_2) \equiv b$, $(s \leqslant t - 1) \equiv b_1$, $(r = t - 1) \equiv b_1$, $cons^-([s..r], \phi) \equiv b_2$.

The remaining cases are defined analogously, and we omit them here.

Observe that the interval bounds s, t in $cons([s..t], \phi) \equiv b$ are always constants with $s \leqslant t$. The formula $\chi \mathsf{U} \phi$ is unfolded into an equivalent disjunction, by

$$\chi \mathsf{U} \phi \quad \equiv \quad \phi \ \vee \ \chi \wedge \bigcirc(\chi \mathsf{U} \phi).$$

We do not deal specially with the bounded quantifiers $\forall, \exists$. They are simply expanded into conjunctions and disjunctions.

10.3.2 Array Translation

This alternative translation avoids the potentially large disjunctive constraints caused by the $\Diamond$ and U operators. The idea is to push disjunctive information into variable domains. Take the example

$$\Diamond\, Q[A, B] = q$$

at the interval $[r..s]$. It can be translated into a single array constraint

$$Q[A, B, x] = q$$

with the fresh variable x whose domain is the set of time points $[r..s]$. We study propagation of such array constraints in Chapter 8.

A complication arises when negation is used: just negating the associated truth value is now incorrect. Consider $\neg\Diamond Q[A, B] = q$ whose translation would be that an $x \in [r..s]$ exists such that ϕ does not hold. We therefore avoid negation. A formula is first transformed into negation normal form (NNF). NNF can be obtained by using some identities, in particular the following on temporal operators.

10.3.1. Fact.

$$\begin{aligned}
\neg\Diamond\phi &= \Box\neg\phi, && (10.1)\\
\neg\Box\phi &= \Diamond\neg\phi, && (10.2)\\
\neg\bigcirc\phi &= \bigcirc\, \mathsf{true} \rightarrow \bigcirc\neg\phi, && (10.3)\\
\neg(\chi \mathsf{U} \phi) &= (\neg\phi) \mathsf{U} (\neg\chi \wedge \neg\phi) \ \vee \ \Box\neg\phi. && (10.4)
\end{aligned}$$

Proof. Identities (10.1) and (10.2) are trivial. For (10.3), note that $\neg\bigcirc\phi$ is always true in the unit interval $[s..s]$ independent of ϕ. The construction $\bigcirc\, \mathsf{true} \rightarrow \psi$ requires then ψ only on intervals $s..t$ with $s < t$.

For (10.4), see the proof given in [Huth and Ryan, 1999, p. 197] of an equivalent identity between formulas in the temporal logic LTL. While the temporal logic that we employ here is not LTL (one essential difference being the interpretation of the $\bigcirc$ operator at unit intervals), the structure of the proof carries over directly in this instance. □

Here is the array translation of NNF formulas. Only the cases different from the unfolding translation are presented, except for negation which is deleted. We only present the translation of 'future' formulas; the case of 'past' formulas is analogous.

$$
\begin{array}{lll}
\mathit{cons}^{+}([s..t], \Box\phi) \equiv b & \text{is} & \mathit{cons}^{+}([s..t], \phi \wedge (\bigcirc\top \rightarrow \bigcirc\Box\phi) \equiv b, \\
\mathit{cons}^{+}([s..t], \Diamond\phi) \equiv b & \text{is} & s \leqslant r,\ r \leqslant t, \\
 & & \mathit{cons}^{+}([r..t], \phi) \equiv b, \\
\mathit{cons}^{+}([s..t], \chi \,\mathsf{U}\, \phi) \equiv b & \text{is} & (b_1 \wedge (b_2 \vee b_3)) \equiv b, \\
 & & s \leqslant r,\ r \leqslant t, \\
 & & \mathit{cons}^{+}([r..t], \phi) \equiv b_1, \\
 & & (s = r) \equiv b_2, \\
 & & s \leqslant u,\ u \leqslant r, \\
 & & (u = r - 1) \equiv b_3, \\
 & & \mathit{cons}^{+}([s..u], \Box\chi) \equiv b_3.
\end{array}
$$

The interval end points s, t in $\mathit{cons}([s..t], \phi)$ are now variables. But observe that the invariant $s \leqslant t$ is always maintained. $\Box\phi$ is unfolded into the formula $\phi \wedge (\bigcirc\, \mathsf{true} \rightarrow \bigcirc\Box\phi)$.

10.3.2. Example. Let us compare the alternative translations using a simplified version of a formula from the earlier navigation domain (p. 163), namely

$$\phi \equiv \quad \Diamond\big(Q[\mathit{ship}, \mathit{buoy}] = \mathsf{E}\ \wedge\ \Diamond Q[\mathit{ship}, \mathit{buoy}] = \mathsf{S}\big).$$

We consider for both translations the sequence of constraints $\mathit{cons}^{+}([1..n], \phi)$, for a constant n. To add readability, we abbreviate

$$
\begin{array}{ll}
\phi_1 \equiv & (Q[\mathit{ship}, \mathit{buoy}] = \mathsf{E}), \\
\phi_2 \equiv & (Q[\mathit{ship}, \mathit{buoy}] = \mathsf{S});
\end{array}
$$

so we inspect $\mathit{cons}^{+}([1..n], \Diamond(\phi_1 \wedge \Diamond\phi_2))$.

Unfolding translation. We obtain in the first translation step

$$
\begin{array}{l}
b_1 \vee \ldots \vee b_n, \\
\qquad \mathit{cons}^{+}([1..n], \phi_1 \wedge \Diamond\phi_2) \equiv b_1, \\
\qquad \vdots \\
\qquad \mathit{cons}^{+}([n..n], \phi_1 \wedge \Diamond\phi_2) \equiv b_n.
\end{array}
$$

Eventually, this becomes

$$
\begin{array}{l}
b_1 \vee \ldots \vee b_n, \\
\qquad (Q[\mathit{ship}, \mathit{buoy}, 1] = \mathsf{E}) \equiv b_1, \\
\qquad b_1 = b_{11} \vee \ldots \vee b_{2n}, \\
\qquad\qquad (Q[\mathit{ship}, \mathit{buoy}, 1] = \mathsf{S}) \equiv b_{11}, \\
\qquad\qquad \vdots \\
\qquad\qquad (Q[\mathit{ship}, \mathit{buoy}, n] = \mathsf{S}) \equiv b_{1n}, \\
\qquad \vdots \\
\qquad (Q[\mathit{ship}, \mathit{buoy}, n-1] = \mathsf{E}) \equiv b_{n-1}, \\
\qquad b_{n-1} = b_{n-1,n-1} \vee b_{n-1,n}, \\
\qquad\qquad (Q[\mathit{ship}, \mathit{buoy}, n-1] = \mathsf{S}) \equiv b_{n-1,n-1}, \\
\qquad\qquad (Q[\mathit{ship}, \mathit{buoy}, n] = \mathsf{S}) \equiv b_{n-1,n}, \\
\qquad (Q[\mathit{ship}, \mathit{buoy}, n] = \mathsf{E}) \equiv b_n, \\
\qquad b_n = b_{nn}, \\
\qquad\qquad (Q[\mathit{ship}, \mathit{buoy}, n] = \mathsf{S}) \equiv b_{nn}.
\end{array}
$$

There are $n + \sum_{i=1}^{n} i = n(n+3)/2$ new Boolean variables, and as many reified equality constraints.

Array translation. We have first

$$1 \leqslant r_1,\ r_1 \leqslant n,\ \mathit{cons}([r_1..n], \phi_1 \wedge \Diamond\phi_2),$$

and

$$1 \leqslant r_1,\ r_1 \leqslant n,\ Q[\mathit{ship}, \mathit{buoy}, r_1] = \mathsf{E},\ r_1 \leqslant r_2,\ r_2 \leqslant n,\ \mathit{cons}([r_2..n], \phi)$$

in the next step. Finally, the result is

$$
\begin{array}{l}
1 \leqslant r_1,\ r_1 \leqslant n,\ Q[\mathit{ship}, \mathit{buoy}, r_1] = \mathsf{E}, \\
r_1 \leqslant r_2,\ r_2 \leqslant n,\ Q[\mathit{ship}, \mathit{buoy}, r_2] = \mathsf{S},
\end{array}
$$

hence two new variables r_1, r_2, two array constraints, and one inequality constraint (if we view unary inequality constraints such as $1 \leqslant r_1$ as simple domain reductions).

As anecdotal evidence that the array translation can lead to substantially better performance, reconsider the full navigation example from Section 10.2.3, page 163. The runtimes in our implementation (to be detailed below) are 2.6 sec with the unfolding translation and 0.4 sec with the array translation, so we observe a speedup roughly of factor 6 in this case. □

10.3.3 Quantification over Objects

The principle of using a variable index in an array constraint instead of an unfolding disjunctive translation can be applied to bounded existential quantification as well. Recall that we defined

$$\exists \mathtt{A} \in \mathcal{O}'.\ \phi(\mathtt{A}) \qquad \text{to abbreviate} \qquad \bigvee_{o \in \mathcal{O}'} \phi(o).$$

Instead of unfolding, we translate this using a new variable, i. e.,

$$\mathit{cons}^{+}([s..t], \exists \mathtt{A} \in \mathcal{O}'.\phi(\mathtt{A})) \equiv b \qquad \text{is} \qquad x_{\mathtt{A}} \in \mathcal{O}',\ \mathit{cons}^{+}([s..t], \phi(x_{\mathtt{A}})) \equiv b.$$

$x_{\mathtt{A}}$ is a fresh object variable ranging over $\mathcal{O}'$.

10.3.3. Example. Consider again the naval navigation domain, this time with a set $\mathcal{S}$ of ships. Let us specify that a state exists in which at least one ship is positioned south-east of the buoy. We formalise

$$\exists s \in \mathcal{S}.\ \Diamond Q[s, \mathit{buoy}] = \mathsf{SE}.$$

The translation at $[1..n]$ into array constraints is the single constraint

$$Q[s, \mathit{buoy}, r] = \mathsf{SE},$$

over two newly introduced variables $s \in \mathcal{S}$ and $r \in [1..n]$. The unfolding translation, in contrast, produces $n \cdot |\mathcal{S}|$ reified constraints. □

10.4 Simulations

By a *qualitative simulation* we mean a finite or infinite sequence

$$\mathcal{PS} \ = \ \langle \mathcal{P}_0, \mathcal{P}_1, \ldots \rangle$$

of CSPs such that for each chosen inter-state constraint $\phi \rightarrow \bigcirc\psi$ we have

– if $\mathcal{PS}$ is finite with u elements, for all $t_0 \in [0 \mathinner{..} u-1], t = t_{\max}$

– if $\mathcal{PS}$ is infinite, for all $t_0 \geqslant 0, t \geqslant t_0 + 1$

the constraint

$$\mathit{cons}([0 \mathinner{..} t_0], \phi) \ \rightarrow \ \mathit{cons}([t_0 + 1 \mathinner{..} t], \psi)$$

is satisfied by the CSP $\bigcup_{i=0}^{t} \mathcal{P}_i$. So at each stage of the qualitative simulation we relate its past (and presence) to its future using the chosen inter-state constraints.

Consider an initial situation $\mathcal{I} = \mathcal{P}_0$ and a final situation $\mathcal{F}_x$ determined by a qualitative array of the form Q_x, where x is a variable ranging over the set of natural numbers (possible time instances). We are interested then in a number of problems. First, we would like to find whether a simulation exists that starts in $\mathcal{I}$ and reaches $\mathcal{F}_t$, where t is the number of steps. If one exists, then we may be interested in computing a shortest one, or in computing all such simulations.

$\boxed{\textsf{Simulate}}$: spatial array Q, state constraints, $t_{\max} \longmapsto$ solution

$t := 0$
$\mathcal{PS} := \langle\rangle$
while $t < t_{\max}$ **do**
  $\mathcal{P}_t :=$ create CSP from Q_t and impose intra-state constraints
  $\mathcal{PS} :=$ append $\mathcal{P}_t$ to $\mathcal{PS}$ and impose inter-state constraints
  $\langle \mathcal{PS}, \mathit{failure}\rangle := \mathsf{prop}(\mathcal{PS})$
  if not *failure* **then**
    $\mathcal{PS}' := \mathcal{PS}$ with final state constraint imposed on $\mathcal{P}_t$
    $\langle \mathit{solution}, \mathit{success}\rangle := \mathsf{solve}(\mathcal{PS}')$
    if *success* **then return** *solution*
  end
  $t := t + 1$
end
return $\varnothing$ // *indicating failure*

Figure 10.4: The simulation algorithm

Simulation algorithm. The algorithm given in Figure 10.4 provides a solution to the first two problems in presence of the non-circularity constraint. We employ here four auxiliary procedures, i. e., create, append, prop and solve, that are used as follows.

- The call to create sets up a new CSP $\mathcal{P}_t$ uniquely determined by the qualitative array Q_t, in which for all objects A,B the domain of the variable Q_t[A,B] equals the set of relations $\mathcal{Q}$. The intra-state constraints are imposed.

- The call to append attaches a CSP to the end of a sequence of CSPs. For each inter-state constraint $\phi \rightarrow \bigcirc\psi$ and $s \in [0..t]$ the constraint $\mathit{cons}^-([0..s], \phi) \rightarrow \mathit{cons}^+([s+1..t], \psi)$ is generated.

- The call prop($\mathcal{PS}$) for a sequence of CSPs $\mathcal{PS} = \mathcal{P}_0, \ldots, \mathcal{P}_t$ performs propagation of the intra-state and inter-state constraints

  If the outcome of the constraint propagation is an inconsistent CSP, the value false is returned in *failure*. An inconsistency can arise if for some value of t the inter-state constraints are unsatisfiable.

- The call solve($\mathcal{PS}$) for a sequence of CSPs of the form $\mathcal{P}_i$ checks if there is a solution to the CSP formed by their union on which the assumed inter-state constraints are imposed. If so, a solution, i. e., an instantiation of the variables of the listed CSPs, and true is returned, otherwise $\langle\varnothing, \mathsf{false}\rangle$.

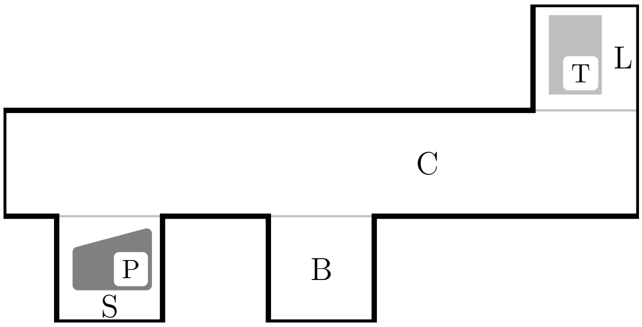

Figure 10.5: A piano movers problem

We use the constant $t_{\max}$ equal to the number of different qualitative arrays, i. e., $t_{\max} = |\mathcal{O}| \cdot (|\mathcal{O}| - 1) \cdot 2^{|\mathcal{Q}|-1}$. If the desired simulation exists, the above algorithm finds a shortest one and outputs it in the variable *solution*.

10.4.1. Example. Consider the following version of the piano movers problem.

> There are three rooms, the living room (L), the study room (S) and the bedroom (B), and the corridor (C). Inside the study room there is a piano (P) and inside the living room a table (T); see Figure 10.5. Move the piano to the living room and the table to the study room assuming that none of the rooms and the corridor are large enough to contain at the same time the piano and the table. Additionally, ensure that the piano and the table at no time will touch each other.

To formalise this problem we first describe the initial situation by means of the following formulas:

$$\begin{aligned}
\phi_0 &\equiv \quad Q[\mathtt{B,L}] = \mathsf{disjoint} \ \wedge \ Q[\mathtt{B,S}] = \mathsf{disjoint} \ \wedge \ Q[\mathtt{L,S}] = \mathsf{disjoint}, \\
\phi_1 &\equiv \quad Q[\mathtt{C,B}] = \mathsf{meet} \ \wedge \ Q[\mathtt{C,L}] = \mathsf{meet} \ \wedge \ Q[\mathtt{C,S}] = \mathsf{meet}, \\
\phi_2 &\equiv \quad Q[\mathtt{P,S}] = \mathsf{inside} \ \wedge \ Q[\mathtt{T,L}] = \mathsf{inside}.
\end{aligned}$$

We assume that initially ϕ_0, ϕ_1, and ϕ_2 hold, i. e., the constraints $cons^-([0\,..\,0], \phi_0)$, $cons^-([0\,..\,0], \phi_1)$ and $cons^-([0\,..\,0], \phi_2)$ are present in the initial situation $\mathcal{I}$.

Below, given a formula ϕ, by an *invariant built out of* ϕ we mean the formula $\phi \rightarrow \bigcirc\Box\phi$. Further, we call a room or a corridor a 'space' and abbreviate the subset of objects $\{\mathtt{B}, \mathtt{C}, \mathtt{L}, \mathtt{S}\}$ to $\mathcal{S}$. We now stipulate as the inter-state constraints the invariants built out of the following formulas:

- the relations between the rooms, and between the rooms and the corridor, do not change:

$$\phi_0 \wedge \phi_1,$$

- at all times, the piano and the table do not fill completely any space:

$$\forall s \in \mathcal{S}.\ (Q[\mathtt{P}, s] \neq \mathsf{equal} \ \wedge \ Q[\mathtt{T}, s] \neq \mathsf{equal})\,,$$

- together, the piano and the table do not fit into any space. More precisely, at each time, at most one of these two objects can be within any space:

$$\forall s \in \mathcal{S}.\ \neg\,(Q[\mathtt{P}, s] \in \{\mathsf{inside}, \mathsf{coveredby}\} \wedge Q[\mathtt{T}, s] \in \{\mathsf{inside}, \mathsf{coveredby}\})\,,$$

- at all time instances the piano and the table do not touch each other:

$$Q[\mathtt{P}, \mathtt{T}] = \mathsf{disjoint}.$$

The final situation is simply captured by the following constraint:

$$Q[\mathtt{P,L}] = \mathsf{inside} \quad \wedge \quad Q[\mathtt{T,S}] = \mathsf{inside}.$$

□

10.5 Implementation and Case Studies

We produced an implementation of the simulation algorithm in Fig. 10.4 and the two translations of temporal formulas to constraints given in Section 10.3. It consists of about 1500 lines of ECL^{i}PS^e code. To test its usefulness we conducted several case studies, of which we report two in the following sections. In both cases, the solutions were found in a few seconds.

10.5.1 Piano Movers Problem

The first report concerns the piano movers problem as formalised in Example 10.4.1. Remarkably, the interaction with our program revealed in the first place that our initial formalisation was incomplete. For example, the program also generated solutions in which the piano is moved not through the corridor but 'through the walls', as it were.

To avoid such solutions we added to the original intra-state constraints the following ones (recall that $\mathcal{S}$ stands for the set $\{\mathtt{B}, \mathtt{C}, \mathtt{L}, \mathtt{S}\}$):

- each space is too small to be 'touched' (*met*) or 'overlapped' by the piano and the table at the same time:

$$\forall s \in \mathcal{S}.\ \neg\,(Q[s, \mathtt{P}] \in \{\mathsf{overlap}, \mathsf{meet}\} \wedge Q[s, \mathtt{T}] \in \{\mathsf{overlap}, \mathsf{meet}\})\,,$$

- if the piano or the table overlaps with one space s, then it also overlaps with some other space s', such that s and s' *touch* each other:

$$\forall s \in \mathcal{S}.\ \forall o \in \{\mathtt{P}, \mathtt{T}\}.$$
$$(Q[s, o] = \mathsf{overlap} \ \rightarrow \ \exists s' \in \mathcal{S}.\ (Q[s', o] = \mathsf{overlap} \wedge Q[s, s'] = \mathsf{meet})),$$

- if the piano overlaps with one space, then it does not *touch* any space, and equally the table:

$$\forall s \in \mathcal{S}.\ \forall o \in \{\mathtt{P}, \mathtt{T}\}.\ \ (Q[s,o] = \mathsf{overlap}\ \rightarrow\ \forall s' \in \mathcal{S}.\ Q[s',o] \neq \mathsf{meet})\,,$$

- both the piano and the table can *touch* at most one space at a time:

$$\forall s, s' \in \mathcal{S}.\ \forall o \in \{\mathtt{P}, \mathtt{T}\}.\\ (Q[s,o] = \mathsf{meet} \land Q[s',o] = \mathsf{meet}\ \rightarrow\ Q[s,s'] = \mathsf{equal})\,.$$

After these additions, our program generated the shortest solution in the form of a simulation of length 12. In this solution the bedroom is used as a temporary storage for the table. Interestingly, the table is not moved completely into the bedroom: at a certain moment it only overlaps with the bedroom.

10.5.2 Phagocytosis

The second example deals with the simulation of phagocytosis; specifically, an amoeba absorbing a food particle. This problem is discussed in [Cui et al., 1992]. We quote:

> Each amoeba is credited with vacuoles (being fluid spaces) containing either enzymes or food which the animal has digested. The enzymes are used by the amoeba to break down the food into nutrient and waste. This is done by routing the enzymes to the food vacuole. Upon contact the enzyme and food vacuoles fuse together and the enzymes merge into the fluid containing the food. After breaking down the food into nutrient and waste, the nutrient is absorbed into the amoeba's protoplasm, leaving the waste material in the vacuole ready to be expelled. The waste vacuole passes to the exterior of the protozoan's (i. e., amoeba's) body, which opens up, letting the waste material pass out of the amoeba and into its environment.

To fit it into our framework, we slightly simplified the problem representation in our approach by not allowing for objects to be added or removed during the simulation.

In this problem, we have six objects, *amoeba*, *nucleus*, *enzyme*, *vacuole*, *nutrient* and *waste*. The initial situation is described by means of the following constraints:

$$Q[\mathit{amoeba}, \mathit{nutrient}] = \mathsf{disjoint},$$
$$Q[\mathit{amoeba}, \mathit{waste}] = \mathsf{disjoint},$$
$$Q[\mathit{nutrient}, \mathit{waste}] = \mathsf{equal}.$$

Further, we have the intra-state constraints

$$
\begin{aligned}
&Q[enzyme, amoeba] = \mathsf{inside},\\
&Q[vacuole, amoeba] \in \{\mathsf{inside}, \mathsf{coveredby}\},\\
&Q[vacuole, enzyme] \in \{\mathsf{disjoint}, \mathsf{meet}, \mathsf{overlap}, \mathsf{covers}\},\\
&Q[nucleus, vacuole] \in \{\mathsf{disjoint}, \mathsf{meet}\},\\
&Q[nucleus, enzyme] \in \{\mathsf{disjoint}, \mathsf{meet}\},\\
&Q[nucleus, amoeba] = \mathsf{inside},
\end{aligned}
$$

and the inter-state constraints,

$$Q[nutrient, amoeba] = \mathsf{meet} \;\rightarrow\; \bigcirc\, Q[nutrient,\, amoeba] = \mathsf{overlap},$$

$$
\begin{aligned}
&Q[nutrient, amoeba] \in \{\mathsf{inside}, \mathsf{coveredby}, \mathsf{overlap}\} \;\rightarrow\\
&\qquad\qquad\qquad\qquad \bigcirc\, Q[nutrient, amoeba] \in \{\mathsf{inside}, \mathsf{coveredby}\}.
\end{aligned}
$$

We model the splitting up of the food into nutrient and waste material by

$$
\begin{aligned}
&Q[nutrient, waste] = \mathsf{equal}\\
&\dot{\rightarrow}\\
&\qquad Q[nutrient, vacuole] = \mathsf{inside} \;\wedge\\
&\qquad Q[enzyme, nutrient] = \mathsf{overlap} \;\wedge\\
&\qquad Q[enzyme, waste] = \mathsf{overlap}\\
&\qquad \dot{\rightarrow}\\
&\qquad\qquad \bigcirc Q[nutrient, waste] = \mathsf{overlap}\\
&\qquad \dot{\vee}\\
&\qquad\qquad \bigcirc Q[nutrient, waste] = \mathsf{equal}\\
&\dot{\vee}\\
&\qquad \bigcirc\, Q[nutrient, waste] \neq \mathsf{equal}.
\end{aligned}
$$

We use here the dotted operators to express *if-then-else*, i. e.

$$a \mathbin{\dot{\rightarrow}} b \mathbin{\dot{\vee}} c \quad \equiv \quad (a \rightarrow b) \;\wedge\; (\neg a \rightarrow c).$$

The final situation is described by means of the constraints

$$
\begin{aligned}
&Q[amoeba, waste] = \mathsf{disjoint},\\
&Q[amoeba, nutrient] \in \{\mathsf{contains}, \mathsf{covers}\}.
\end{aligned}
$$

Our program generated a solution that consists of 9 steps.

10.6 Final Remarks

The most common approach to qualitative simulation is the one discussed in [Kuipers, 1994, chapter 5]. For a recent overview see [Kuipers, 2001]. It is based on a qualitative differential equation model (QDE) in which one abstracts from the usual differential equations by reasoning about a finite set of symbolic values (called *landmark values*). The resulting algorithm, called *QSIM*, constructs the tree of possible evolutions by repeatedly constructing the successor states. During this process, CSPs are generated and solved.

This approach is best suited to simulate evolution of physical systems. A standard example is a simulation of the behaviour of a bath tub with an open drain and constant input flow. The resulting constraints are usually equations between the relevant variables and lend themselves naturally to a formalisation using CLP(FD), see [Bratko, 2001, chapter 20] and [Bandelj et al., 2002]. The limited expressiveness of this approach was overcome in [Brajnik and Clancy, 1998], where branching time temporal logic was used to describe the relevant constraints on the possible evolutions (called 'trajectories' there). This leads to a modified version of the *QSIM* algorithm in which model checking is repeatedly used.

Our approach is inspired by the qualitative spatial simulation studied in [Cui et al., 1992], the main features of which are captured by the composition table and the neighbourhood relation discussed in Example 10.2.1. The distinction between the intra- and inter-state constraints is introduced there, however the latter only link the consecutive states in the simulation. The simulation algorithm of [Cui et al., 1992] generates a complete tree of all 'evolutions'.

In contrast to [Cui et al., 1992], our approach is constraint-based. This allows us to repeatedly use constraint propagation to prune the search space in the simulation algorithm. Further, by using more complex inter-state constraints, defined by means of temporal logic, we can express substantially more sophisticated forms of behaviour. Our approach can be easily implemented on top of a constraint programming system, using a relation variable model.

Simulation in our approach subsumes a form of planning. In this context, we mention the related work [Lopez and Bacchus, 2003] in the area of planning which shows the benefits of encoding planning problems as CSPs and the potential with respect to solving efficiency. Also related is the TLplan system where planning domain knowledge is described in temporal logic [Bacchus and Kabanza, 2000]. The planning system is based on incremental forward-search, so temporal formulas are just unfolded a step at a time, in contrast to the translation into constraints in our constraint-based system.

Finally, [Faltings, 2000] discusses how a qualitative version of the piano movers problem can be solved using an approach to qualitative reasoning based on topological inference and graph-theoretic algorithms. Our approach is simpler in that it does not rely on any results on topology apart of a justification of the composition table given in Figure 9.2.

Chapter 11

Final Remarks

11.1 Summary

One ideal of constraint programming is to be declarative: modelling the problem should be independent of the solution algorithm. The rule-based approach to constraint propagation which we examined in this thesis is entirely within this line; rules express constraint propagation declaratively.

Propagation by rules. In the three chapters following the introductory part, we dealt with generic issues concerning rule-based propagation. We developed theoretically well-founded techniques and implemented them as concrete tools.

We started with showing how a generic fixpoint computation algorithm can be instantiated to the improved scheduling algorithm R for constraint propagation rules. The central observation is this: the information that the condition of a rule succeeds entails *more* than just that it is now correct to apply the rule body. It also may state something about other rules: truth of their conditions, or relevance of their body. Such information can be used to accelerate the fixpoint computation, in particular by shrinking the set of participating rules. Most usefully, these connections can be pre-computed. Moreover, since fixpoint computations for the purpose of constraint propagation are called in successive rounds, rules removed in one round need not be reconsidered in subsequent rounds. The set of propagation rules shrinks as search and propagation progresses.

The removal of a rule during a fixpoint computation in the way just described indicates a 'local redundancy'. Rather similarly, a rule can also be (globally) redundant. This is the case if the common fixpoints of a rule set do not change if the rule is removed. Surprisingly, the relation between redundancy and what we call 'local redundancy' here is not straightforward: a redundant rule is not always 'locally redundant', nor does the inverse statement hold true.

Empirical examinations of the two techniques, i. e., using the R rule scheduler that dynamically reduces the set of involved functions, and removing redundant rules from a rule set as a pre-process, showed that both can be very useful. They also appear to be orthogonal as far as improving the efficiency is concerned. Both techniques are thus of relevance for efficient rule-based constraint propagation.

We also dealt with the origin of constraint propagation rules. In contrast to existing automatic rule generation methods, we considered incremental rule generation, based on modifying and combining rules instead of relying on the constraint definition. For a concrete rule type, the membership rules, we examined various cases of incremental rule generation, with a focus on the associated propagation.

Applications. The usefulness of these techniques was evaluated in concrete applications. Our study of the automatic test pattern generation problem led to three models with different multi-valued logics. These logics naturally capture structural properties of the problem. In turn, the constraints arising from these models were directly applicable to the rule-based propagation techniques we developed.

Our approach to modal satisfiability checking was similar in the sense that we also applied a non-Boolean logic to express structural and heuristic information. Constraint propagation could in part be dealt with by the techniques mentioned above such as the R algorithm and redundancy removal during preprocessing. However, several complex constraints required manually designed and implemented propagation rules. The resulting constraint-based modal-SAT solver exhibited a performance competitive with an approach relying on an advanced, purely propositional solver.

Array constraints are not directly amenable to rule-based propagation as discussed above. However, the problem of their propagation can be tackled with a rule-based design approach. For two local consistencies, we derived concrete rules from generic rule templates embodying the desired propagation, and turned them subsequently into propagation algorithms.

In the area of qualitative spatial representation and reasoning, information is naturally organised in arrays. We discussed an alternative to the standard constraint-based model of qualitative space, and we argued for its advantages, which eminently lie in the ease of modelling several spatial aspects. Such models use simple constraints to which our rule-based propagation techniques are directly applicable.

Our approach to constraint-based qualitative simulation builds on the alternative model for (static) qualitative spatial reasoning, and in this way shows its flexibility. We used temporal logic formulas to represent knowledge about change, which yielded compact formalisations. We provided translations of such formulas into constraints, which allows to link them by propagation.

11.2 Outlook

We gather some future research issues involving rule-based propagation.

One question concerning propagation rules for simple constraints is how succinct this approach generally is. Consider membership rules, for example. While it is in principle possible to generate all GAC-enforcing membership rules for any given constraint, clearly their number will often be large. An intuitive argument for this is that the same atomic piece of information, a non-solution, occurs multiple times in different rules. It would be useful to better understand how the number of GAC-enforcing membership rules, with and without redundancy removal, depends on the number of solutions or also the structure of the constraint.

An alternative, interesting road for rule-based propagation is indicated in Sections 7.4.2 (p. 115) and 8.2.1, where we manually derived rules for complex, 'global' constraints from intensional definitions. It would be instructive to formalise in detail what derivation steps were taken, in order to ultimately automate these rule derivations. Such an approach could be based on a constraint definition in the shape of a logical formula, from which logical implications are derived that in turn are viewed as constraint propagation rules.

A further question regards the issue of control. In the R scheduler, we left open in which order rules are chosen and applied. The rule selection strategy clearly influences the length of the computation, however.

Finally, we must raise the issue of how efficient, in terms of complexity, propagation algorithms based on constraint propagation rules can in principle be. Instead of the localised view on a problem taken by each rule individually, a global view may afford more useful structuring of the information [Maher, 2002a]. This point comes to mind, for example, when one regards 'global' constraints whose propagation algorithms rely on graph-theoretic methods and of which the underlying propagation rules are not directly obvious.

Such observations may suggest a necessary departure from the concept of a constraint propagation rule in the pure sense of Def. 2.2.1. It may not necessarily require us to leave the rule-based paradigm, however; for example, see [Ganzinger and McAllester, 2002] where classical algorithms with optimal complexity are described in rule form.

To conclude this dissertation: in our view the rule-based paradigm provides a useful and elegant approach to constraint propagation, and, while much remains to be explored, we think it has great potential.

Bibliography

[Abdennadher and Frühwirth, 2002] Abdennadher, S. and Frühwirth, T. (2002). Using program analysis for integration and optimization of rule-based constraint solvers. In *Proc. of Journées Francophones de Progr. Logique et Progr. par Contraintes (JFPLC'02).*

[Abdennadher and Frühwirth, 2003] Abdennadher, S. and Frühwirth, T. (2003). Integration and optimization of rule-based constraint solvers. In *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'03).*

[Abdennadher et al., 2002] Abdennadher, S., Krämer, E., Saft, M., and Schmauss, M. (2002). JACK: A Java constraint kit. In *Proc. of 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP'01)*, volume 64 of *ENTCS*. Elsevier.

[Abdennadher and Rigotti, 2001] Abdennadher, S. and Rigotti, C. (2001). Using confluence to generate rule-based constraint solvers. In *Proc. of 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 127–135. ACM.

[Abdennadher and Rigotti, 2002] Abdennadher, S. and Rigotti, C. (2002). Automatic generation of rule-based solvers for intentionally defined constraints. *International Journal on Artificial Intelligence Tools*, 11(2):283–302.

[Abdennadher and Rigotti, 2004] Abdennadher, S. and Rigotti, C. (2004). Automatic generation of rule-based constraint solvers over finite domains. *ACM Transactions on Computational Logic*, 5(2):177–205.

[Allen, 1983] Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843.

[Apt, 1998] Apt, K. R. (1998). A proof theoretic view of constraint programming. *Fundamenta Informaticae*, 34(3):295–321.

[Apt, 1999] Apt, K. R. (1999). The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210.

[Apt, 2000] Apt, K. R. (2000). The role of commutativity in constraint propagation algorithms. *ACM Transactions on Programming Languages and Systems*, 22(6):1002–1036.

[Apt, 2003] Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

[Apt and Brand, 2003] Apt, K. R. and Brand, S. (2003). Schedulers for rule-based constraint programming. In *Proc. of ACM Symposium on Applied Computing (SAC'03)*, pages 14–21. ACM Press.

[Apt and Monfroy, 2001] Apt, K. R. and Monfroy, E. (2001). Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 1(6):713–750.

[Areces et al., 2000] Areces, C., Gennari, R., Heguiabehere, J., and de Rijke, M. (2000). Tree-based heuristics in modal theorem proving. In *Proc. of 14th European Conference on Artificial Intelligence (ECAI'00)*, pages 199–203. IOS Press.

[Azevedo, 2003] Azevedo, F. (2003). *Constraint Solving over Multi-valued Logics – Application to Digital Circuits*, volume 91 of *Frontiers of Artificial Intelligence and Applications*. IOS Press.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., editors (2003). *The Description Logic HandBook*. Cambridge University Press.

[Baader et al., 1992] Baader, F., Franconi, E., Hollunder, B., Nebel, B., and Profitlich, H.-J. (1992). An empirical analysis of optimization techniques for terminological representation systems, or making KRIS get a move on. In *Proc. of 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 270–281.

[Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press.

[Bacchus and Kabanza, 2000] Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116.

[Bandelj et al., 2002] Bandelj, A., Bratko, I., and Suc, D. (2002). Qualitative simulation with CLP. In *Proc. of 16th International Workshop on Qualitative Reasoning (QR'02).*

[Beacham et al., 2001] Beacham, A., Chen, X., Sillito, J., and van Beek, P. (2001). Constraint programming lessons learned from crossword puzzles. In *Proc. of 14th Canadian Conference on Artificial Intelligence*, pages 78–87.

[Beldiceanu, 2000a] Beldiceanu, N. (2000a). Global constraints as graph properties on a structured network of elementary constaints of the same type. In [Dechter, 2000], pages 52–66.

[Beldiceanu, 2000b] Beldiceanu, N. (2000b). Global constraints as graph properties on a structured network of elementary constaints of the same type. Technical report, Swedish Institute of Computer Science.

[Beldiceanu and Contejean, 1994] Beldiceanu, N. and Contejean, E. (1994). Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123.

[Bennett, 1998] Bennett, B. (1998). Determining consistency of topological relations. *Constraints*, 2:213–225.

[Bessière, 1996] Bessière, C. (1996). Random uniform CSP generator. Available at `www.lirmm.fr/~bessiere/generator.html`,.

[Blackburn et al., 2001] Blackburn, P., de Rijke, M., and Venema, Y. (2001). *Modal Logic*. Cambridge University Press.

[Bordeaux and Monfroy, 2002] Bordeaux, L. and Monfroy, E. (2002). Beyond NP: Arc-consistency for quantified constraints. In [Hentenryck, 2002], pages 371–386.

[Borovanský et al., 1998] Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., and Ringeissen, C. (1998). An overview of ELAN. In Kirchner, C. and Kirchner, H., editors, *Proc. of 2nd International Workshop on Rewriting Logic and its Applications*, volume 15 of *ENTCS*. Elsevier.

[Brajnik and Clancy, 1998] Brajnik, G. and Clancy, D. (1998). Focusing qualitative simulation using temporal logic: theoretical foundations. *Annals of Mathematics and Artificial Intelligence*, 22:59–86.

[Brand, 2001a] Brand, S. (2001a). Constraint propagation in presence of arrays. *Joint Bulletin of the Novosibirsk Computing Center and Institute of Informatics Systems*, 16:99–110.

[Brand, 2001b] Brand, S. (2001b). Sequential automatic test pattern generation by constraint programming. In *CP'01 Post Conference Workshop Modelling and Problem Formulation (FORMUL'01)*.

[Brand, 2003] Brand, S. (2003). A note on redundant rules in rule-based constraint programming. In O'Sullivan, B., editor, *Recent Advances in Constraints*, volume 2627 of *LNAI*, pages 109–120. Springer.

[Brand, 2004] Brand, S. (2004). Relation variables in qualitative spatial reasoning. In *Proc. of 20th German Annual Conference on Artificial Intelligence (KI'04)*, LNAI. Springer. Accepted for publication.

[Brand and Apt, 2005] Brand, S. and Apt, K. (2005). Schedulers and redundancy for rule-based constraint programming. *Theory and Practice of Logic Programming*. To appear.

[Brand et al., 2004] Brand, S., Gennari, R., and de Rijke, M. (2004). Constraint methods for modal satisfiability. In Apt, K. R., Fages, F., Rossi, F., Szeredi, P., and Váncza, J., editors, *Recent Advances in Constraints*, volume 3010 of *LNAI*, pages 66–86. Springer.

[Brand and Monfroy, 2003] Brand, S. and Monfroy, E. (2003). Deductive generation of constraint propagation rules. In Giavitto, J.-L. and Moreau, P.-E., editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier.

[Bratko, 2001] Bratko, I. (2001). *PROLOG Programming for Artificial Intelligence*. International Computer Science Series. Addison-Wesley, third edition.

[Brglez et al., 1989] Brglez, F., Bryan, D., and Kozminski, K. (1989). Combinational profiles of sequential benchmark circuits. In *Proc. of IEEE Int. Symposium on Circuits and Systems*, pages 1929–1934. ISCAS'89 Benchmark available at `www.cbl.ncsu.edu/CBL_Docs/iscas89.html`.

[Cardelli and Gordon, 2000] Cardelli, L. and Gordon, A. D. (2000). Anytime, anywhere. modal logics for mobile ambients. In *Proc. of 27th ACM Symposium on Principles of Programming Languages*. ACM Press.

[Carlson et al., 1995] Carlson, B., Carlsson, M., and Janson, S. (1995). The implementation of AKL(FD). In Lloyd, J. W., editor, *Proc. of International Symposium on Logic Programming (ILPS'95)*, pages 227–241. MIT Press.

[Carlson et al., 1994] Carlson, B., Haridi, S., and Janson, S. (1994). AKL(FD) – A concurrent language for FD programming. In Bruynooghe, M., editor, *Proc. of International Symposium on Logic Programming (ILPS'94)*, pages 521–538. MIT Press.

[Caseau et al., 2002] Caseau, Y., Josset, F.-X., and Laburthe, F. (2002). CLAIRE: Combining sets, search and rules to better express algorithms. *Theory and Practice of Logic Programming*, 2(6).

[Castro, 1998] Castro, C. (1998). Building constraint satisfaction problem solvers using rewrite rules and strategies. *Fundamenta Informaticae*, 34(3):263–293.

[Cheng, 1996] Cheng, K. T. (1996). Gate-level test generation for sequential circuits. *ACM Transactions on Design Automation of Electronic Systems*, 1(4):405–442.

[Choi et al., 2003] Choi, C. W., Lee, J. H. M., and Stuckey, P. J. (2003). Propagation redundancy in redundant modelling. In [Rossi, 2003], pages 229–243.

[Cicerone and Felice, 2004] Cicerone, S. and Felice, P. D. (2004). Cardinal directions between spatial objects: the pairwise-consistency problem. *Information Sciences*, 164:165–188.

[Codognet and Diaz, 1996] Codognet, P. and Diaz, D. (1996). Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226.

[Cohn and Hazarika, 2001] Cohn, A. G. and Hazarika, S. M. (2001). Qualitative spatial representation and reasoning: An overview. *Fundamenta Informaticae*, 46(1-2):1–29.

[Cristani, 1999] Cristani, M. (1999). The complexity of reasoning about spatial congruence. *Journal of Artificial Intelligence Research*, 11:361–390.

[Cui et al., 1992] Cui, Z., Cohn, A. G., and Randell, D. A. (1992). Qualitative simulation based on a logical formalism of space and time. In Rosenbloom, P. and Szolovits, P., editors, *Proc. of 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 679–684. AAAI Press.

[Dao et al., 2002] Dao, T.-B.-H., Lallouet, A., Legtchenko, A., and Martin, L. (2002). Indexical-based solver learning. In [Hentenryck, 2002], pages 541–555.

[Dechter, 2000] Dechter, R., editor (2000). *Proc. of 6th International Conference on Principles and Practice of Constraint Programming (CP'00)*, volume 1894 of *LNCS*. Springer.

[Dechter, 2003] Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

[Dechter and van Beek, 1997] Dechter, R. and van Beek, P. (1997). Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308.

[Dincbas et al., 1988] Dincbas, M., Hentenryck, P. V., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. (1988). The constraint logic programming language CHIP. In for New Generation Computer Technology (ICOT), I., editor, *Proc. of International Conference on Fifth Generation Computer Systems*, volume 2, pages 693–702. Springer.

[Egenhofer, 1991] Egenhofer, M. J. (1991). Reasoning about binary topological relations. In Günther, O. and Schek, H.-J., editors, *Proc. of 2nd International Symposium on Large Spatial Databases (SSD'91)*, volume 525 of *LNCS*, pages 143–160. Springer.

[Faltings, 2000] Faltings, B. (2000). Using topology for spatial reasoning. In *Proc. of 8th International Symposium on Artificial Intelligence and Mathematics (AI&M'00)*.

[Faltings and Macho-Gonzalez, 2002] Faltings, B. and Macho-Gonzalez, S. (2002). Open constraint satisfaction. In [Hentenryck, 2002], pages 356–370.

[Fitting, 1992] Fitting, M. C. (1992). Many-valued modal logics II. *Fundamenta Informaticae*, XVII:55–74.

[Forgy, 1981] Forgy, C. L. (1981). OPS5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, Dept. of Computer Science.

[Frank, 1992] Frank, A. U. (1992). Qualitative spatial reasoning about distance and directions in geographic space. *Journal of Visual Languages and Computing*, 3:343–373.

[Freksa and Zimmermann, 1992] Freksa, C. and Zimmermann, K. (1992). On the utilization of spatial structures for cognitively plausible and efficient reasoning. In *Proc. of IEEE International Conference on Systems, Man, and Cybernetics*, pages 18–21. IEEE.

[Freuder, 1978] Freuder, E. C. (1978). Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966.

[Frühwirth, 1998] Frühwirth, T. (1998). Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138.

[Frühwirth and Abdennadher, 2003] Frühwirth, T. and Abdennadher, S. (2003). *Essentials of Constraint Programming.* Springer.

[Ganzinger and McAllester, 2002] Ganzinger, H. and McAllester, D. (2002). Logical algorithms. In [Hentenryck, 2002], pages 148–162.

[Gent et al., 2000] Gent, I., van Maaren, H., and Walsh, T., editors (2000). *SAT 2000. Highlights of Satisfiability Research in the Year 2000.*, volume 63 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

[Gerevini and Renz, 2002] Gerevini, A. and Renz, J. (2002). Combining topological and size constraints for spatial reasoning. *Artificial Intelligence*, 137(1-2):1–42.

[Gervet, 1997] Gervet, C. (1997). Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244.

[Giunchiglia and Sebastiani, 2000] Giunchiglia, F. and Sebastiani, R. (2000). Building decision procedures for modal logics from propositional decision procedures. the case study of modal $K(m)$. *Information and Computation*, 162(1-2):158–178.

[Goyal and Egenhofer, 1997] Goyal, R. K. and Egenhofer, M. J. (1997). The direction-relation matrix: A representation of direction relations for extended spatial objects. In *Proc. of UCGIS Annual Assembly and Summer Retreat*.

[Haarslev and Möller, 2002] Haarslev, V. and Möller, R. (2002). RACER. Accessed via `kogs-www.informatik.uni-hamburg.de/∼race/`.

[Halpern et al., 2001] Halpern, J. Y., Harper, R., Immerman, N., Kolaitis, P. G., Vardi, M. Y., and Vianu, V. (2001). On the unusual effectiveness of logic in computer science. *The Bulletin of Symbolic Logic*, 7:213–236.

[Haralick and Elliott, 1980] Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313.

[Harrenstein et al., 2002] Harrenstein, B. P., van der Hoek, W., Meyer, J.-J. C., and Witteveen, C. (2002). On modal logic interpretations of games. In *Proc. of 15th European Conference on Artificial Intelligence (ECAI'02)*.

[Hentenryck, 1989] Hentenryck, P. V. (1989). *Constraint Satisfaction in Logic Programming*. MIT Press.

[Hentenryck, 2002] Hentenryck, P. V., editor (2002). *Proc. of 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*. Springer.

[Hentenryck et al., 1999] Hentenryck, P. V., Lustig, I., Michel, L., and Puget, J.-F. (1999). *The OPL optimization programming language*. MIT Press.

[Hentenryck et al., 1992] Hentenryck, P. V., Simonis, H., and Dincbas, M. (1992). Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1-3):113–159.

[Heuerding and Schwendimann, 1996] Heuerding, A. and Schwendimann, S. (1996). A benchmark method for the propositional modal logics K, KT, S4. Technical Report IAM-96-015, University of Bern.

[Hnich, 2003] Hnich, B. (2003). *Function Variables for Constraint Programming.* PhD thesis, Dept. of CS, Uppsala University.

[Holzbaur, 2002] Holzbaur, C. (2002). Source-to-source transformation for constraint handling rules. In *Proc. of Workshop on Functional and Logic Programming (WFLP'02).*

[Holzbaur et al., 2001] Holzbaur, C., de la Banda, M. J. G., Jeffery, D., and Stuckey, P. J. (2001). Optimizing compilation of Constraint Handling Rules. In Codognet, P., editor, *Proc. of 17th International Conference on Logic Programming (ICLP'01)*, volume 2237 of *LNCS*, pages 74–89. Springer.

[Hooker et al., 2000] Hooker, J. N., Ottosson, G., Thorsteinsson, E. S., and Kim, H.-J. (2000). A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review, Special Issue on Artifical Intelligence and Operations Research*, 15(1):11–30.

[Horrocks, 2002] Horrocks, I. (2002). FaCT. Accessed via `www.cs.man.ac.uk/~horrocks/FaCT/`.

[Horrocks et al., 2000] Horrocks, I., Patel-Schneider, P., and Sebastiani, R. (2000). An analysis of empirical testing for modal decision procedures. *Logic Journal of the IGPL*, 8(3):293–323.

[Hustadt and Schmidt, 1997] Hustadt, U. and Schmidt, R. A. (1997). On evaluating decision procedures for modal logic. In *Proc. of IJCAI-97*, pages 202–207.

[Huth and Ryan, 1999] Huth, M. R. A. and Ryan, M. D. (1999). *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press.

[ILOG, 2000] ILOG (2000). *OPL Studio 3.* ILOG S.A.

[Isli, 2003] Isli, A. (2003). Combining cardinal direction relations and relative orientation relations in qualitative spatial reasoning. Technical report, University of Hamburg, Dept. of Informatics.

[Isli, 2004] Isli, A. (2004). Combining cardinal direction relations and other orientation relations in QSR. In *Proc. of 8th International Symposium on Artificial Intelligence and Mathematics (AI&M'04).*

[Isli and Cohn, 2000] Isli, A. and Cohn, A. G. (2000). A new approach to cyclic ordering of 2D orientations using ternary relation algebras. *Artificial Intelligence*, 122(1-2):137–187.

[Jaffar and Lassez, 1987] Jaffar, J. and Lassez, J.-L. (1987). Constraint logic programming. In *Proc. of 14th Annual ACM Symposium on Principles of Programming Languages (POPL'87).*

[Jaffar and Maher, 1994] Jaffar, J. and Maher, M. J. (1994). Constraint logic programming: A survey. *Journal of Logic Programming*, 19 & 20:503–582.

[Katsirelos and Bacchus, 2001] Katsirelos, G. and Bacchus, F. (2001). GAC on conjunctions of constraints. In Walsh, T., editor, *Proc. of 7th International Conference on Principles and Practice of Constraint Programming (CP'01)*, volume 2239 of *LNCS*, pages 610–614. Springer.

[Kirchner and Ringeissen, 1998] Kirchner, C. and Ringeissen, C. (1998). Rule-based constraint programming. *Fundamenta Informaticae*, 34(3):225–262.

[Kleene, 1952] Kleene, S. C. (1952). *Introduction to Metamathematics.* Van Nostrand, New York.

[Kuipers, 1994] Kuipers, B. (1994). *Qualitative reasoning: modeling and simulation with incomplete knowledge.* MIT Press.

[Kuipers, 2001] Kuipers, B. (2001). *Encyclopedia of Physical Science and Technology*, chapter Qualitative simulation, pages 287–300. Academic Press, third edition.

[Kumar, 1992] Kumar, V. (1992). Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44.

[Lamma et al., 1999] Lamma, E., Milano, M., and Mello, P. (1999). Reasoning on constraints in CLP(FD). *Journal of Logic Programming*, 38(1):93–110.

[Ligozat, 1998] Ligozat, G. (1998). Reasoning about cardinal directions. *Journal of Visual Languages and Computing*, 9(1):23–44.

[Lloyd, 1987] Lloyd, J. (1987). *Foundations of Logic Programming.* Springer, 2nd extended edition.

[Lopez and Bacchus, 2003] Lopez, A. and Bacchus, F. (2003). Generalizing GraphPlan by formulating planning as a CSP. In *Proc. of International Joint Conference on Artificial Intelligence (IJCAI'03).*

[M. Carlsson et al., 2004] M. Carlsson et al. (2004). *SICStus Prolog User's Manual*. Swedish Institute of Computer Science.

[Mackworth, 1977] Mackworth, A. K. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1):118–126.

[Mackworth and Freuder, 1985] Mackworth, A. K. and Freuder, E. C. (1985). The complexity of some polynomial network algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74.

[Maher, 2002a] Maher, M. J. (2002a). Analysis of a global contiguity constraint. In *Proc. of 4th Workshop on Rule-based Constraint Reasoning and Programming (RCoRP'02)*.

[Maher, 2002b] Maher, M. J. (2002b). Propagation completeness of reactive constraints. In Stuckey, P. J., editor, *Proc. of 18th International Conference on Logic Programming (ICLP'02)*, volume 2401 of *LNCS*, pages 148–162. Springer.

[Markey et al., 2002] Markey, N., Laroussinie, F., and Schnoebelen, P. (2002). Temporal logic with forgettable past. In *Proc. of 17th IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 383–392.

[Marriot and Stuckey, 1998] Marriot, K. and Stuckey, P. J. (1998). *Programming with Constraints*. MIT Press.

[Martelli and Montanari, 1982] Martelli, A. and Montanari, U. (1982). An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282.

[Marx, 2004] Marx, M. (2004). XPath with conditional axis relations. In *Proc. of International Conference on Extending Database Technology*.

[Mohr and Masini, 1988] Mohr, R. and Masini, G. (1988). Good old discrete relaxation. In Kodratoff, Y., editor, *Proc. of European Conference on Artificial Intelligence (ECAI'88)*, pages 651–656. Pitman publishers.

[Montanari, 1974] Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132.

[MSPASS, 2001] MSPASS (2001). MSPASS v. 1.0.0t.1.2.a. Accessed via `www.cs.man.ac.uk/~schmidt/mspass/`.

[Muth, 1976] Muth, P. (1976). A nine-valued circuit model for test generation. *IEEE Transactions on Computers*, 25(6):630–636.

[Pan et al., 2002] Pan, G., Sattler, U., and Vardi, M. Y. (2002). BDD-based decision procedures for K. In *Proc. of 18th Conference on Automated Deduction (CADE'02)*, pages 16–30. Springer.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational complexity.* Addison Wesley Longman.

[Patel-Schneider, 2002] Patel-Schneider, P. F. (2002). DLP. Accessed via `www.bell-labs.com.user/pfps/dlp/`.

[Pistore and Traverso, 2001] Pistore, M. and Traverso, P. (2001). Planning as model checking for extended goals in non-deterministic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'01).*

[Randell et al., 1992a] Randell, D. A., Cohn, A. G., and Cui, Z. (1992a). Computing transitivity tables: A challenge for automated theorem provers. In *Proc. of 11th Conference on Automated Deduction (CADE'92)*, volume 607 of *LNAI*. Springer.

[Randell et al., 1992b] Randell, D. A., Cui, Z., and Cohn, A. G. (1992b). A spatial logic based on regions and connection. In Nebel, B., Rich, C., and Swartout, W. R., editors, *Proc. of 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pages 165–176. Morgan Kaufmann.

[Rao and Georgeff, 1998] Rao, A. and Georgeff, M. (1998). Decision procedures for BDI logics. *Journal of Logic and Computation*, 8:293–342.

[Renz and Nebel, 2001] Renz, J. and Nebel, B. (2001). Efficient methods for qualitative spatial reasoning. *Journal of Artificial Intelligence Research*, 15:289–318.

[Ringeissen and Monfroy, 2000] Ringeissen, C. and Monfroy, E. (2000). Generating propagation rules for finite domains via unification in finite algebras. In Apt, K. R., Kakas, A. C., Monfroy, E., and Rossi, F., editors, *New Trends in Constraints*, volume 1865 of *LNAI*, pages 150–172. Springer.

[Rossi, 2003] Rossi, F., editor (2003). *Proc. of 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *LNCS*. Springer.

[Roth, 1966] Roth, J. P. (1966). Diagnosis of automata failure: A calculus and a method. *IBM Journal of Research and Development*, 10(4):278–291.

[Saraswat, 1993] Saraswat, V. A. (1993). *Concurrent Constraint Programming.* MIT Press.

[Shiva, 1988] Shiva, S. G. (1988). *Introduction to logic design.* Scott, Foresman/Little.

[Simonis, 1989] Simonis, H. (1989). Test generation using the constraint logic programming language CHIP. In Levi, G. and Martelli, M., editors, *Proc. of 6th International Conference on Logic Programming (ICLP'89)*, pages 101–112. MIT Press.

[Skiadopoulos and Koubarakis, 2001] Skiadopoulos, S. and Koubarakis, M. (2001). Composing cardinal direction relations. In Jensen, C., Schneider, M., Seeger, B., and Tsotras, V., editors, *Proc. of 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD'01)*, volume 2121 of *LNCS*, pages 371–386. Springer.

[Smolka, 1995] Smolka, G. (1995). The Oz programming model. In van Leeuwen, J., editor, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer.

[Tacchella, 1999] Tacchella, A. (1999). *SAT system description. In *Collected Papers from the International Description Logics Workshop, CEUR*.

[TANCS, 2000] TANCS (2000). TANCS: Tableaux non-classical systems comparison. Accessed via `www.dis.uniroma1.it/~tancs/`.

[Tsang, 1993] Tsang, E. (1993). *Foundations of Constraint Satisfaction.* Academic Press.

[Tsang, 1987] Tsang, E. P. K. (1987). The consistent labeling problem in temporal reasoning. In Forbus, K. S. H., editor, *Proc. of 6th National Conference on Artificial Intelligence (AAAI'87)*, pages 251–255. AAAI Press.

[Wallace et al., 1997] Wallace, M. G., Novello, S., and Schimpf, J. (1997). ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200.

[Walsh, 2000] Walsh, T. (2000). SAT v CSP. In [Dechter, 2000], pages 441–456.

[Walsh, 2003] Walsh, T. (2003). Consistency and propagation with multiset constraints: A formal viewpoint. In [Rossi, 2003], pages 724–738.

Index