Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XVII Ciclo – 2004

Declarative constraint modelling and specification-level reasoning

Toni Mancini

UNIVERSITÀ DEGLI STUDI DI ROMA "LA SAPIENZA"

DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA

XVII CICLO - 2004

Toni Mancini

Declarative constraint modelling and specification-level reasoning

Thesis Committee		Reviewers
Prof. Marco Cadoli	(Advisor)	Prof. Pierre Flener
Prof. Giorgio Ausiello		Prof. Miroslaw Truszczynski
Prof. Diego Calvanese		

AUTHOR'S ADDRESS:
Toni Mancini
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
E-MAIL: `tmancini@dis.uniroma1.it`
WWW: `http://www.dis.uniroma1.it/~tmancini`

Abstract

Declarative problem modelling is becoming the new challenge for constraint programming. However, in current systems, the efficiency of computation may be strongly affected in several ways. First of all, different but equivalent models for the same problem do, in general, exist, and choosing one of them can make the difference. Secondly, the role played by the search strategy to be followed when traversing the search space is fundamental. Nowadays, it is the user that chooses, based on his/her experience, the best model for a problem, and the most appropriate search strategy to solve it.

On the other hand, different techniques have been proposed in order to improve the efficiency of computation, e.g., symmetry breaking, abstraction, addition of useful implied constraints, etc. However, many of these techniques apply, to the best of our knowledge, after the commitment to the instance, and very little work has been done at the symbolic level of the problem specification.

Moreover, many languages for constraint programming actually exhibit a clear separation between problem specifications and instances, and many properties of constraint problems that are amenable to be optimized rely on the problem structure, not on the particular instance considered. Hence, it becomes clear how the detection of these properties at the instance level, when the problem structure has been almost completely hidden, may easily become unconvenient.

In this thesis we discuss how problem specifications can be regarded as logical formulae, and how different forms of reasoning can be performed, in order to reformulate them with the goal of improving the efficiency of the solving process. We deal with different forms of reasoning on problem specifications, highlighting those constraints that can be safely ignored in a first stage, and then efficiently reinforced, detecting and breaking symmetries, detecting and exploiting functional dependencies, and recognizing specifications that can take advantage of Knowledge Compilation. Moreover, we show how these forms of reasoning can be practically made by computer, by exploiting current ATP technology. Experimental results show how reasoning on problem specifications can significantly improve performances when using different classes of solvers.

The second topic covered in this thesis is the design of highly declarative languages for constraint modelling, suitable also in industrial environments. In particular, we propose *NP-Alg* and ConSql, obtained as extensions of relational algebra and SQL, respectively. The main motivation under the proposal of such languages is that, in commercially available packages for constraint programming, there is no true integration between the data definition and the constraint modelling and programming languages, with the main disadvantage of providing poor levels of data integrity. Proposed languages, instead, can be actually considered as second-order query languages for relational databases, providing strong integration between constraint modelling and programming and up-to-date technology for storing and querying data. We believe that this feature can allow a wider diffusion of the declarative constraint modelling paradigm in industrial environments, permitting a very strong integration with the information system of the enterprise.

Acknowledgements

I would like to thank all the people who helped me during the precious years of my PhD. First of all I am profoundly grateful to my advisor and friend Marco Cadoli, who taught me a lot, guided and encouraged me continuously, and dedicated me so much of his time and attention.

Next, I wish to thank all other members of the thesis committee and external reviewers: Giorgio Ausiello, Diego Calvanese, Pierre Flener, and Miroslaw Truszczynski, who gave me precious comments and indications that much improved the quality of this work. I'm also very grateful to the colleagues I worked with in these years, in particular the (not yet mentioned) coauthors of the papers I have written: Lucas Bordeaux, Giuseppe De Giacomo, Massimo Mecella, and Giuseppe Santucci.

I am also very grateful to the other members of the AI group of the Dipartimento di Informatica e Sistemistica of the Università "La Sapienza" di Roma, and in particular to Luigia Carlucci Aiello, Maurizio Lenzerini, and Marco Schaerf.

MIUR (Italian Ministry for Instruction, University, and Research) FIRB project ASTRO (Automazione dell'Ingegneria del Software basata su Conoscenza), and the COFIN project "Design and development of a software system for the specification and efficient solution of combinatorial problems, based on a high-level language, and techniques for intensional reasoning and local search" allowed me to attend conferences and to use equipment necessary for my research.

Last but not least, I wish to thank my family and all my friends for encouragement they gave me.

Contents

Chapter 1

Introduction

The paradigm of declarative programming is becoming very attractive for solving different classes of problems. In particular, Constraint Programming (CP) has emerged as a winning approach for, among the others, combinatorial, scheduling, planning, and resource allocation problems, since the high declarativeness of problem specifications may coexist with the increasing efficiency of state-of-the-art solvers. To this end, a constraint problem specification may definitively be viewed as a *program*, that is given in a highly declarative language, and in which users state *what* conditions a solution must satisfy, rather than *how* it can be computed. Ideally, this second issue is left to the system.

The CP paradigm has shown great advantages over "traditional" (i.e., imperative) programming: problem specifications can be highly declarative, by describing the problem in terms of a search space and a set of conditions a generic element of it has to satisfy to be a solution of the problem. Moreover, CP programs are usually much shorter, readable and reusable than traditional ones. Finally, they present advantages also with respect to other declarative paradigms, since several techniques have been developed with the aim of increasing their efficiency (e.g., constraint propagation, local consistency notions, etc.)

Even if CP can be actually considered only as a fruitful application of already well known techniques, the novel element was the appearing (in the '70s and '80s) of several new languages, having built-in constraint primitives. Very well known languages and systems for CP are, e.g., PROLOG II (35), PROLOG III (36), CLP(R) (76), CHIP (89), BNR-PROLOG (103), L_λ (97), etc. Most of them rely on the subclass of CP called *Constraint logic programming* (CLP), since they were derived from logic programming languages as PROLOG (cf. (75) for a survey).

Nowadays, CP is becoming a very promising approach also in industrial applications, since the maturity of the field and the developed techniques allow to deal with problem instances of larger size. AMPL (57), OPL (124), GAMS (27) and XPRESS$^{\text{MP}}$[1], are examples of commercial packages and libraries for programming with constraints. Academic interests are now also directed into the realization of deductive systems whose semantics is based on non-monotonic formalisms. Significant proposals in this context are the systems DLV (85), SMODELS (118) and the language PS (47).

[1]cf. `http://www.dashoptimization.com`.

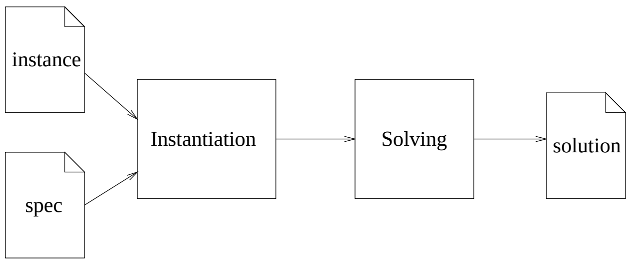

Figure 1.1: Two-level architecture for the solution of combinatorial problems

State-of-the-art specification languages have been realized by extending the CP methodology essentially in two ways:

1. By adding more linguistic constructors to specify the problem, e.g., by considering the possibility of expressing various form of quantification, or the so called "higher-order constraints", in which arbitrary relations can be part of the expressions (as e.g., in OPL), and by allowing a neat separation between the problem specification and the instance, thus allowing for reusable, maintainable and extendible specifications not dependent on particular input data;

2. By adding a real "programming" component (in the sense of the traditional programming languages), by means of which it is possible to generate, combine, and elaborate constraints.

As subsequent sections will show, in thesis we focus on the first point, providing languages for high level declarative modelling of combinatorial and constraint problems, and techniques that allow such models to be more efficiently evaluable.

1.1 Declarative modelling languages for combinatorial problems

Even if the CP paradigm naturally yields a certain degree of declarativeness in problem specifications, current CP systems often rely on additional information provided by the user (heuristics, choice of the algorithm, etc.) to improve their efficiency. This leads to a lowering of the degree of declarativeness. Moreover, in some existent systems, the problem specification is mixed with the input instance, resulting in a specification that is less readable, reusable, maintainable, and extendable.

To this end, a current promising approach is to provide languages with a clean separation between problem specification and input instance (e.g., AMPL, OPL, XPRESS^{MP}, ESRA (55), EaCL (122), NP-Spec (15), NP-Alg and ConSql, cf. Chapter 8). User provides the generic definition of the problem, and an instance of it separately. In this way, new instances of the same problem can be solved without modifications to the specification. The general architecture these systems rely on is depicted in Figure 1.1: the specification is firstly instantiated against the instance, and only then a particular solver is invoked.

The major benefit of separation between specification and instance is the decoupling of the solver from the specification. Ideally, the programmer can focus only on the specification of the problem, without committing *a priori* to a specific solver. In fact, some systems are able to translate –at the request of the user– a specification in various formats, suitable for different solvers. Probably the best known of such systems is AMPL, which offers an interface for the specification (also called *model*) of a Mathematical Programming problem, each instance of which can be, at the request of the user, translated and solved by one of several solvers for integer and/or linear programming, among them CPLEX (72), LANCELOT[2], MINOS[3], and many others.[4]

Some systems go one step further, and offer a limited form of *reasoning* on the specification, with the goal of choosing the most appropriate solver for a problem. As an example, the OPL system checks whether a specification contains only linear constraints and objective function, and in this case invokes an integer linear programming solver (typically very efficient); otherwise, it uses a constraint programming solver.

With these languages, much emphasis is given to the qualities of problem specifications viewed as *software artifacts*. Readability, extendability, maintainability, verifiability and reusability are viewed as key issues like correctness and efficiency. *Declarative problem modelling* is thus coming up as the next goal for CP systems (cf., e.g., (58)), by attempting to provide languages in which the search space specification and the constraints are evidently distinguished and highly-level modelled, and no procedural aspect is included.

However, in state-of-the-art systems, the efficiency of computation can be highly affected in several ways. First of all, different but equivalent formulations for the same problem do, in general, exist, and choosing one of them can make the difference. To this end, many techniques and several approaches have been proposed in order to modify a given constraint problem into an equivalent form, with the goal of reducing the solving time. Such techniques include, e.g., the addition of new constraints, e.g., symmetry-breaking or implied constraints, in order to reduce the size of the search space, or the opposite strategy of deleting (abstracting) some constraints in order to obtain a simplified problem whose solutions can be used to compute, in an efficient way (in some cases without further search), valid solutions of the original one. All of the approaches listed above lead to different and widely independent (even if correlated) research fields.

The second point that can greatly affect the efficiency of the solving engine is the choice of the search strategy to be applied, and the heuristic to be used to order variables and domain values during branches. Many systems actually require that the user defines a search strategy for the problem at hand, thus greatly loosing in declarativeness. However some of them, e.g., OPL, even if allow the specification of a search strategy, do not require it, thus allowing the user to focus only on the combinatorial aspects of the problem. Nonetheless, even if the default search strategy performs quite well in many cases, there are situations where a smart reasoning on the problem specification permits to infer much better strategies. A good example is given by specifications where functional dependencies among variables exist, either because of a precise modelling choice (e.g., in order to increase modularity, readabil-

[2]cf. `http://www.cse.clrc.ac.uk/nag/lancelot/lancelot.shtml`.
[3]cf. `http://www.sbsi-sol-optimize.com/`.
[4]cf. `http://www.ampl.com`.

ity, or other qualities of the specification), or because of intrinsic properties of the modelled problem. If some of the variables are recognized to be dependent on the others, the declarative constraint program provided by the user can be transformed by synthesizing a search strategy that exploits such dependencies, e.g., by avoiding branches on those variables (cf. Section 5.4).

Although much research has been done on these aspects, all techniques proposed in the literature in order to optimize the solving process either apply at the instance level, or are reformulations of a specific constraint problem, obtained as the output of a true design process: it is the designer that chooses, based on his/her experience, the best formulation for a problem for the particular solver used.

On the other hand, our research aims to a more ambitious long-term goal: the *automated reformulation* of a purely declarative constraint problem specification, into a form that is more efficiently evaluable for the solver at hand. In particular, in this thesis make a first step towards this direction, investigating the problem of reformulating declarative specifications in several and complementary directions.

A second important question that arises when dealing with highly declarative languages, is what is the class of problems that can be formulated with them. In general, a formal characterization of the computational properties of specification languages is indeed very useful both when designing new ones, and when choosing a particular one for the class of problems to be solved. Unfortunately, many of the available CP languages don't have a formal characterization of their computational properties (e.g., OPL). In Chapter 8 we propose *NP-Alg*, a modelling language based on relational algebra, and provide a formal analysis of its computational properties, showing what is its expressive power, as well as its data and combined complexity.

1.2 Existential second order logic as an abstract modelling language

The style used for the specification of a combinatorial problem varies a lot among different languages for constraint programming. As already claimed in Chapter 1, in this thesis, rather than considering procedural encodings such as those obtained using libraries (in, e.g., C++ or PROLOG), we focus on highly declarative languages.

Again, the syntax varies a lot among such languages: AMPL, OPL, XPRESS$^{\text{MP}}$, ESRA and GAMS allow the representation of constraints by using algebraic expressions, while DLV, SMODELS, and NP-SPEC are rule-based languages, more specifically extensions of datalog. Anyway, from an abstract point of view, all such languages are extensions of *existential second-order logic* (ESO) over finite databases, where the existential second-order quantifiers and the first-order formula represent, respectively, the *guess* and *check* phases of the constraint modelling paradigm. In particular, even if all such languages have a richer syntax and more complex constructs, in all of them it is possible to embed ESO queries, and the other way around is also possible, as long as only finite domains are considered. Hence, as we show in the remainder of this section, ESO can be considered as the formal basis for virtually all available languages for constraint modelling, being able to represent all search problems in the complexity class NP (51; 105). Intuitively, the relationship between ESO and available modelling languages is similar to that holding between Turing machines or assembler, and high-level programming languages.

To this end, in this thesis we use ESO logic to model specifications, for at least two reasons:

1. We believe that studying the simplified scenario is a mandatory starting point for our investigations, because of the simplicity of the model;

2. We believe that our results can serve as a basis for reformulating specifications written in richer languages, thus greatly improving interoperability. To this end, in the following sections we show how our techniques can be applied to specifications given in high-level languages, e.g., AMPL and OPL, and prove their effectiveness by experimental results using state-of-the-art solvers.

Moreover, we point out that a logic-based approach has also been successfully adopted in the '80s to study the query optimization problem for relational databases. Analogously to our approach, the query optimization problem has been studied in a formal way using first-order logic (cf., e.g., (3; 30; 111; 80)). In a later stage, the theoretical framework has been translated into rules for the automated rewriting of queries expressed in languages and systems used in real world.

Formally, an ESO formula ψ is of the form:

$$\exists \vec{S}\ \phi(\vec{S}, \vec{R}), \tag{1.1}$$

where $\vec{R}$ denotes a fixed set of relational symbols not including equality "=", $\vec{S} = \{S_1, \ldots, S_h\}$ denote variables ranging over relational symbols (of various arities) distinct from those in $\vec{R} \cup \{=\}$, and ϕ is a function-free first-order formula containing occurrences of relational symbols from $\vec{R} \cup \vec{S} \cup \{=\}$. The symbol "=" is always interpreted as identity.

By Fagin's theorem (51; 105), any collection $\mathbf{D}$ of finite databases over $\vec{R}$ is recognizable in NP time if and only if it is defined by an ESO formula of the kind (1.1). A database D is in $\mathbf{D}$ if and only if there exists a list of relations $\Sigma_1, \ldots, \Sigma_h$ (matching the list of relational variables $S_1, \ldots, S_h$) that, along with D, satisfies formula (1.1), i.e., such that

$$(D, \Sigma_1, \ldots, \Sigma_h) \models \phi.$$

The tuples of $\Sigma_1, \ldots, \Sigma_h$ must take elements from the *Herbrand universe* of D, i.e., the set of constant symbols occurring in it, plus those occurring in ϕ.

Being capable to express all problems in NP, ESO can be definitively considered as an abstract modelling language for constraint problems. In particular, an ESO specification describing a search problem π is a formula ψ_π of the kind (1.1), where $\vec{R}$ is the relational schema for every input instance.

Coherently with all state-of-the-art systems, an instance $\mathcal{I}$ to problem π is given as a *relational database* over the schema $\vec{R}$, i.e., as an extension for all relations in $\vec{R}$. All constants appearing in the database are *uninterpreted*, i.e., they don't have a specific meaning. Existentially quantified predicates, i.e., those in the set $\vec{S} = \{S_1, \ldots, S_n\}$ (of given arities) are called *guessed*, and their possible extensions (with tuples with components in the Herbrand universe) encode points in the search space for problem

```
int+ n_nodes = ...;
int+ n_colors = ...;
range nodes 1..n_nodes;
range colors 1..n_colors;

struct edge {
   nodes start;
   nodes end;
};
{edge} edges = ...;

var colors coloring[nodes];

solve {
   forall (e in edges : e.start <> e.end) {
    (coloring[e.start] <> coloring[e.end]);
   };
};
```

Figure 1.2: OPL specification for the Graph k-coloring problem.

π. Formula ψ_π correctly encodes problem π if, for every input instance $\mathcal{I}$, a bijective mapping exists between solutions to $\pi(\mathcal{I})$ and extensions of predicates in $\vec{S}$ that verify $\phi(\vec{S},\mathcal{I})$:

$$\text{For each instance } \mathcal{I}\text{:}\quad \Sigma \text{ is a solution to } \pi(\mathcal{I}) \quad \Longleftrightarrow \quad \{\Sigma,\mathcal{I}\} \models \phi(\vec{S},\mathcal{I}).$$

In other words, solutions for problem π on input instance $\mathcal{I}$ are those extensions for guessed predicates in $\vec{S}$ with components in the Herbrand domain that make $\phi(\vec{S},\mathcal{I})$ true.

It is worthwhile to note that, when a specification is instantiated against an input database, a CSP in the sense of (44) is obtained.

Example 1.2.1 (Graph k-coloring (62, Prob. GT4, p. 191)). *Given an undirected graph and a positive integer k as input, this problem amounts to decide whether it is possible to give each of its nodes one out of k colors, in such a way that adjacent nodes (not including self-loops) are never colored the same way. The problem is well-known to be* NP*-complete for any $k \geq 3$.*

Figure 1.2 shows an OPL *encoding of this problem. An input instance is given by providing the number of nodes, the number of colors, and an extension for relation edge which encodes the edges of the input graph. A solution to this problem is an extension for the* `coloring[]` *array (denoted with the keyword* `var`*) such that every edge (not including self-loops) links nodes with different colors (*good coloring *constraint).*

The same problem can be specified in ESO by, e.g., the following formula over relations $node(\cdot)$, $edge(\cdot,\cdot)$, and $color(\cdot)$, listing the graph nodes, edges and the available colors, respectively. In particular, relation $color(\cdot)$ will have exactly k tuples. We also assume that $node(\cdot)$ and $color(\cdot)$ have no tuples in common.

```
param n_nodes;
param n_colors integer, > 0;
set NODES := 1..n_nodes;
set EDGES within NODES cross NODES;
set COLORS := 1..n_colors;

# Coloring of nodes as a 2-ary predicate
var Coloring {NODES,COLORS} binary;

s.t. CoveringAndDisjointness {x in NODES}:
     # nodes have exactly one color
     sum {c in COLORS} Coloring[x,c] = 1;

s.t. GoodColoring {(x,y) in EDGES, c in COLORS}:
     # nodes linked by an edge have diff. colors
     Coloring[x,c] + Coloring[y,c] <= 1;
```

Figure 1.3: AMPL specification for Graph k-coloring.

$$\begin{aligned}
\exists Col \quad & \forall XY \quad Col(X,Y) \rightarrow node(X) \wedge color(Y) \ \wedge && (1.2)\\
& \forall X\ \exists Y \quad node(X) \rightarrow Col(X,Y) \ \wedge && (1.3)\\
& \forall XYZ \quad Col(X,Y) \wedge Col(X,Z) \rightarrow Y = Z \ \wedge && (1.4)\\
& \forall XYZ \quad X \neq Y \wedge Col(X,Z) \wedge Col(Y,Z) \rightarrow \neg edge(X,Y). && (1.5)
\end{aligned}$$

Constraint (1.2) *forces components of the guessed predicate* $Col(\cdot,\cdot)$ *to be of the right types, while* (1.3) *and* (1.4) *impose every node to be assigned exactly one color. Constraint* (1.5) *is the good coloring constraint.*

As it can be observed, OPL variables play the same role of guessed predicates in the ESO specification. One of the main differences between ESO and richer modelling languages like OPL is the absence of types and constructs for specifying functions. However, the above example shows *(i)* how functions from a given domain to a given codomain can be modelled by guessing relations and imposing additional constraints over them, i.e., (1.2–1.4), and *(ii)* how bounded integers can be represented (in the simplest way) in unary form. We further discuss how to model functions, integers, and orderings in ESO later in this section.

The additional constraints needed in an ESO specification to model functions may suggest, to a first look, that ESO is actually too far away from languages provided by commercial systems. However, this is not true in general. As an example, Figure 1.3 shows the specification of the Graph k-coloring problem in AMPL (57), which admits only linear constraints: as it can be observed, it is very similar to that of Example 1.2.1, in that covering and disjointness constraints must be explicitly stated.

In constraint problems, domains for variables are often finite (cf. the domain for the `coloring[]` array in Figure 1.2 and those for the Coloring predicate in Figure 1.3). Furthermore, in many circumstances, the size of these domains is "small", and independent on the particular instance. In such cases, there exist alternative, but equivalent, ESO specifications. In general, given an n-ary guessed predicate P with one of the arguments over a domain of size m (independent on the instance), we can always *unfold* it into m $(n-1)$-ary guessed predicates $P_1, \ldots, P_m$, one for each such values. The remaining of the specification must be unfolded accordingly.

As an example, in the following we give an ESO specification for the Graph 3-coloring problem, in which the binary guessed predicate $Col(\cdot,\cdot)$ is unfolded according to the second argument, hence obtaining 3 monadic guessed predicates, R, G, B, one for each colors (red, green, blue).

Example 1.2.2 (Graph 3-coloring). *An ESO specification of the Graph* 3*-coloring problem with monadic guessed predicates is as follows:*

$$\begin{array}{lll}
\exists RGB & \forall X & R(X) \vee G(X) \vee B(X) \ \wedge \qquad (1.6)\\
 & \forall X & R(X) \rightarrow \neg G(X) \ \wedge \qquad (1.7)\\
 & \forall X & R(X) \rightarrow \neg B(X) \ \wedge \qquad (1.8)\\
 & \forall X & B(X) \rightarrow \neg G(X) \ \wedge \qquad (1.9)\\
 & \forall XY & X \neq Y \wedge R(X) \wedge R(Y) \rightarrow \neg edge(X,Y) \wedge \qquad (1.10)\\
 & \forall XY & X \neq Y \wedge G(X) \wedge G(Y) \rightarrow \neg edge(X,Y) \wedge \qquad (1.11)\\
 & \forall XY & X \neq Y \wedge B(X) \wedge B(Y) \rightarrow \neg edge(X,Y), \qquad (1.12)
\end{array}$$

Clauses (1.6) and (1.7-1.9) are obtained as the unfolding of, respectively constraints (1.3) *and* (1.4)*, while (1.10–1.12) are the unfolding of* (1.5)*. It is interesting to observe that constraint* (1.2) *can be thrown away, since the Herbrand domain of the latter specification is made only by the graph nodes.*

Referring to the graph in Figure 2.1, the Herbrand universe is the set $\{a, b, c, d, e\}$, the input database has only one relation, i.e., edge, which has five tuples (one for each edge). Formula ψ is satisfied if R, G, and B are assigned to, e.g., the following relations (cf. Figure 2.1, right):

$$R \ \boxed{d} \qquad G \ \boxed{\begin{array}{c} a \\ e \end{array}} \qquad B \ \boxed{\begin{array}{c} b \\ c \end{array}}$$

Another simple example of ESO specification is the following.

Example 1.2.3 (Not-all-equal Sat (62, Prob. LO3)). *In this NP-complete problem the input is a propositional formula in CNF, and the question is whether it is possible to assign a truth value to all the variables in such a way that the input formula is satisfied, and that every clause contains at least one literal whose truth value is false. We assume that the input formula is encoded by the following relations (an alternative format for the input is described in Section 7.2):*

- *$inclause(\cdot,\cdot)$; tuple $\langle l, c\rangle$ is in inclause if and only if literal l is in clause c;*
- *$l^+(\cdot,\cdot)$ (resp. l^-); a tuple $\langle l, v\rangle$ is in l^+ (resp. l^-) if and only if l is the positive (resp. negative) literal relative to variable v, i.e., v itself (resp. $\neg v$);*

- *$var(\cdot)$, containing the set of propositional variables occurring in the formula;*
- *$clause(\cdot)$, containing the set of clauses of the formula.*

An ESO specification for this problem is as follows (T and F represent the set of variables whose truth value is true and false, respectively):

$$
\begin{aligned}
&\exists TF\ \forall X\ var(X) \leftrightarrow T(X) \lor F(X) \quad \land && (1.13)\\
&\quad \forall X\ \neg\,(T(X) \land F(X)) \quad \land && (1.14)\\
&\quad \forall C\ clause(C) \rightarrow \\
&\qquad \big[\exists L\ inclause(L,C) \land \forall V\ \big(l^{+}(L,V) \rightarrow T(V)\big) \land \big(l^{-}(L,V) \rightarrow F(V)\big)\,\big] \ \land && (1.15)\\
&\quad \forall C\ clause(C) \rightarrow \\
&\qquad \big[\exists L\ inclause(L,C) \land \forall V\ \big(l^{+}(L,V) \rightarrow F(V)\big) \land \big(l^{-}(L,V) \rightarrow T(V)\big)\,\big]. && (1.16)
\end{aligned}
$$

Constraints (1.13–1.14) force every variable to be assigned exactly one truth value; moreover, (1.15) *forces the assignment to be a model of the formula, while* (1.16) *leaves in every clause at least one literal whose truth value is false.*

In what follows, the set of tuples from the Herbrand universe taken by guessed predicates will be called their *extension* and denoted with $ext()$. As an example, $ext(G) = \{a, e\}$ in Example 1.2.2. The symbol $ext()$ will be used also for any first-order formula with one free variable. An interpretation will be sometimes denoted as the aggregate of several extensions.

As already argued in Example 1.2.1, even if ESO does not have built-in constructs for functions, bounded integers, and ordering, these can be defined by applying additional constraints on the extensions of guessed predicates. Here, we show how opportune short-hands can be defined in order to deal with these issues in the general case, increasing the readability and analyzability of problem specifications written in ESO. We remark that functions, bounded integers, and ordering do not add any expressive power to the language. In particular, some interesting expressions that can be defined (formal definitions are delayed to Appendix A) are:

- $typedRelation(F^{(d+r)}, D^{(d)}, R^{(r)})$ evaluates to `true` if and only if guessed predicate F is a relation from tuples in relation D (domain) to those in R (codomain).
- $function_{d,r}(F^{(d+r)})$ evaluates to `true` if and only if guessed predicate F is mono-valued, i.e., a function. First d arguments denote its domain, last r its codomain. In other words, for every d-tuple $\langle x_1, \ldots, x_d\rangle$ that occurs as the first d components of any tuple in F, there exists at most one r-tuple $\langle y_1, \ldots, y_r\rangle$ such that $\langle x_1, \ldots, x_d, y_1, \ldots, y_r\rangle$ belongs to F.
- $total(F^{(d+r)}, D^{(d)})$ evaluates to `true` if and only if guessed predicate F is such that for every d-tuple $\langle x_1, \ldots, x_d\rangle$ in D there exists at least one r-tuple $\langle y_1, \ldots, y_r\rangle$ such that $\langle x_1, \ldots, x_d, y_1, \ldots, y_r\rangle$ belongs to F. That is, F is a total relation on the domain D.
- $surgective(F^{(d+r)}, R^{(r)})$ evaluates to `true` if and only if guessed predicate F is such that for every r-tuple $\langle y_1, \ldots, y_r\rangle$ in R there exists at least one d-tuple

$\langle x_1, \ldots, x_d \rangle$ such that $\langle x_1, \ldots, x_d, y_1, \ldots, y_r \rangle$ belongs to F. That is, F is a surgective relation on the codomain R.

- $\mathit{injective}_{d,r}(F^{(d+r)})$ evaluates to `true` if and only if guessed predicate F is such that for every r-tuple $\langle y_1, \ldots, y_r \rangle$ occurring as the last components of a tuple in F there exists at most one d-tuple $\langle x_1, \ldots, x_d \rangle$ such that $\langle x_1, \ldots, x_d, y_1, \ldots, y_r \rangle$ belongs to F.

- $\mathit{bijective}(F^{(d+r)}, D^{(d)}, R^{(r)})$ evaluates to `true` if and only if guessed predicate F is a bijective function from tuples in D to those in R.

- $\mathit{linearOrder}(\mathit{ORDER}^{(2k)}, N^{(k)})$ evaluates to `true` if and only if guessed predicate ORDER correctly encodes a successor relation over elements in relation N, i.e., a 1-1 correspondence with the interval $[1, |N|]$.

By guessing a linear order among tuples of a relation, e.g., N, we have *bounded integers* up to $|N|$, and can further encode in ESO formulae that compare two integers and compute arithmetic operations over them, as long as the results of these operations are bounded. Details of these encodings are given in Appendix A. It is worth noting that the possibility to encode bounded integers and arithmetics in ESO has interesting consequences. In particular, it is known that many solvers, e.g., SAT-based, or those based on extensions of datalog, e.g., NP-Spec, dlv, do not have built-in integers and arithmetics, and this is a major obstacle for the convenient modelling of a great variety of problems. Definitions given in Appendix A can be straightforwardly rewritten to let these languages deal with these issues.

In Constraint Programming, problems of interest usually belong to the complexity class NP. This is why we restrict ourselves to the existential fragment of second-order logic when defining a modelling language for CP. However, many reasoning problems in Artificial Intelligence reside in different complexity classes, e.g., Σ_2^{P}, Π_2^{P}, etc., higher in the Polynomial Hierarchy. To this end, when dealing with these more complex problems (cf., e.g., Chapter 7), we need larger fragments of second-order logic. In general, we remind that full, i.e., with arbitrary sequences of second-order quantifiers, second-order logic over finite databases is able to express all problems in the complexity class PSPACE.

1.3 Reasoning on the specification

Reasoning on constraint problems in order to change their formulations has been proven to be of fundamental importance in order to speed-up the solving process. To this end, different approaches have been proposed in the literature, among them *symmetry detection* and *breaking*, *abstraction* and *simplification* of constraints, generation of useful *implied constraints*, and the use of *redundant models*. We briefly describe these and other approaches below.

Symmetries arise very frequently in constraint problems, and are one of the major bottlenecks for a backtracking solver, making the size of the search space for the problem at hand much larger than necessary. In fact, if a point in the search space has been proven to be not a solution, all the symmetric points can be immediately excluded from search, since they cannot be solutions at all. On the other hand, when a solution is found, all symmetric points in the search space are guaranteed to be solutions as well, and no additional check is needed for them. Symmetry detection and breaking has been attacked in different ways by the research community. We delay the description of the main approaches in Section 4.1.

A second approach to problem reformulation is *abstraction*, i.e., the simplification of some of the constrains in order to find more efficient reformulations of the original problem (cf. e.g., (43; 66; 59)). In particular, in (59) a form of abstraction is presented, that may require backtracking for finding solutions of the original problem. In Chapters 2 and 3 we present a different approach to constraint simplification, called *safe delay*, which is backtracking free.

A different strategy is the generation of *implied constraints*, i.e., constraints that are not explicitly stated, but that follow from the others. Explicitly stating (some of the) implied constraints can lead some algorithms for constraint programming to make better propagation, obtaining better pruning of the search space. Several works have been proposed in this context, among them (121) and the the system CGRASS (60), which automatically breaks symmetries and generate useful implied constraints.

Another technique to obtain better propagation is the use of *redundant models*, i.e., multiple viewpoints of the search space for the problem at hand synchronized by channelling constraints (33; 53; 129). Each constraint is then expressed under the most convenient viewpoint, e.g., the one that allows better propagation. Also the search strategy can exploit the different viewpoints, hence leading to a better choice of the variables to branch on (71).

Finally, other approaches to problem reformulation exist. As an example, in (131) it is shown how to translate an instantiated CSP into its Boolean form, which is useful for finding different reformulations, while in (34) the proposed approach is to generate a conjunctive decomposition of an instantiated CSP, by localizing independent subproblems. Some attempts have also been done in formalizing and systematizing the process of selecting the best formulation for a problem (10), and in deriving alternative CSP formulations by automatically refining an abstract problem specification (5).

However, many of the approaches described above either are designed for a specific constraint problem, or act at the instance level, and very little work has been performed at the level of problem specification.

Indeed, many of the properties of constraint problems amenable to optimizations, e.g., symmetries, existence of “useful” implied constraints, of constraints that can be simplified, or existence of multiple viewpoints, strongly depend on the problem structure, and not on the particular input instance considered. Hence, their recognition at the instance level, where the problem structure has been almost completely hidden, may easily become not convenient and expensive.

Moreover, as already observed in Section 1.1, almost all state-of-the-art systems for constraint modelling and programming exhibit a clear separation between problem specification and input instances. Hence, it becomes very attractive, from a methodological point of view, detecting and exploiting those properties of constraint problems amenable to optimizations that derive from the problem structure, at the symbolic level, before binding the specification to a particular instance. This leads to the automated reformulation of specifications.

In general, given a declarative problem specification π, the task of reformulating π is to compute a new specification π' (possibly integrated with additional information about the search strategy to be used) such that:

1. For every input instance $\mathcal{I}$, $\pi(\mathcal{I})$ is equivalent to $\pi'(\mathcal{I})$, or, at least, solutions of $\pi(\mathcal{I})$ can be computed in a efficient way starting from solutions of $\pi'(\mathcal{I})$;

2. π' is more efficiently evaluable than π.

Of course the effectiveness of a reformulation may depend both on the problem and on the solver, but this does not rule out the possibility to find reformulations that are good for all solvers (or for solvers of a certain class, e.g., SAT-based, linear solver, or local-search based ones).

A similar approach has already been widely used in other contexts, e.g. that of relational DBMSs. There, the query provided by the user is preprocessed in order to find the so called *best execution plan*. In many cases, the query reformulation does not account for the current database extension. As an example, a widely used heuristic is to make selections as soon as possible to decrease the number of accesses to disk (cf., e.g., (2)). The rewritten query is then used to retrieve the result.

On the other hand, even if there have been some attempts to deal with specifications rewriting (e.g., into instances of SAT (26)), at the moment no general methodology has been developed for CP systems.

In general, reasoning at the specification level can be done in many different ways, e.g., by modifying the search space shape, adding, removing (i.e., abstracting) or modifying constraints, detecting and exploiting structural properties like symmetries or functional dependencies, or recognizing particular classes of problems that are efficiently solvable in different ways. Obviously, once such a specification preprocessing is made by the system, all classical techniques developed for CP at the instance level can yet be used.

Unfortunately, reformulating a constraint problem specification is a difficult task in general: as argued in Section 1.2, a specification is essentially a formula in second-order logic, and it is well known that the equivalence problem is undecidable already in the first-order case (9). For this reason, research must focus on controlled and restricted forms of reformulation.

In this thesis we present several and complementary techniques for reformulating declarative constraint specifications, as Section 1.4 outlines. The intended overall approach is to find criteria for reformulating specifications that can be automatically performed by the system. To this end, some of the techniques presented in this thesis rely on syntactic properties of the specifications. We remind that the syntactic approach has been used with success also in other works, e.g., in (69) for characterizing polynomial and NP-complete sub-cases of existential second-order logic over graphs, and in (83) for characterizing approximable NP optimization problems.

However, some other reformulation techniques proposed here intrinsically rely on semantic (and undecidable) properties of the specification. Indeed, in these cases we show how, in practical circumstances, automated tools, e.g., first-order theorem provers, can be effectively used to make the required reasoning (cf. Chapter 6).

1.4 Outline of the thesis, and summary of results

In this thesis, we focus on the reformulation problem for constraint problem specifications, expressed as second-order formulae, in different directions.

In particular, in Chapter 2 we show how to highlight some of the constraints in a specification that can be safely ignored in a first step of the solving process and efficiently enforced afterwards, i.e., the so called *safe-delay constraints*. This technique permits a simplification of the original specification, yet guaranteeing that every solution of the rewritten problem can be transformed in a valid solution of the original one. Furthermore, the transformation is guaranteed to be performed in polynomial time, i.e., without further search. We present several examples of specifications that exhibit such kind of constraints, like Graph coloring (cf. Example 1.2.2), Not-all-equal Sat (cf. Example 1.2.3), Job-shop scheduling, and others, and experimental results that prove the effectiveness of the approach for different classes of solvers. In Chapter 3 we consider an extension of reformulation by safe-delay for a class of permutation problems. We show that in this case, more complex modifications of the remaining constraints of the specification to be reformulated may be needed to ensure correctness. We consider well known permutation problems, like Hamiltonian path, Permutation flow shop and Tiling, and show how the technique applies to these problems. Experimental results show the effectiveness of the reformulation.

Chapter 4 deals with the well-known symmetry detection problem. As already argued in Section 1.3, symmetries arise frequently in problem specifications, and are a major bottleneck for many solvers. However, differently with other approaches, we focus on those symmetries that depend on the problem structure, and recognize them at level of the specification. To this end, we formally characterize the so called *uniform value symmetries* in problem specifications, and reduce the problem of detecting them into that of checking the equivalence of two first-order formulae. Moreover, when symmetries have been found, additional constraints, the so called *symmetry-breaking formulae* can be added to the specification itself in order to break them. We formally define what properties a formula must satisfy in order to be symmetry-breaking, in terms of semantic properties of logical formulae, and present examples and experimental results about the effectiveness of breaking symmetries in this way.

Chapter 5 focuses on another major issue that degrades solvers performances, i.e., the presence, in declarative problem specifications, of functional dependencies among different guessed predicates. This happens whenever state information, e.g., intermediate results, has to be maintained in order to evaluate constraints and/or the objective function, or because of precise modelling choices, e.g., the so called *redundant modelling*, that consists in having multiple viewpoints of the search space (synchronized by channeling constraints), in order to express each constraint under the most effective one. We give a precise formalization of functional dependencies among guessed predicates, and show how the problem of recognizing a functional dependence can be reduced to checking whether a first-order formula is valid. Moreover, once functional dependencies have been detected, opportune search strategies can be automatically synthesized in order to exploit them, hence greatly reducing the size of the explored search space. We give examples of problem specifications that exhibit functional dependencies and show how they can be automatically exploited. Furthermore, experimental results show how the addition of the synthesized search strategies greatly enhances the performances of state-of-the art solvers.

The proposed techniques for detecting symmetries and functional dependencies of problem specifications reduce to checking semantic properties of first-order formulae. Moreover, in previous chapters, we prove that, in general, these tasks are undecidable. In chapter 6, we investigate the use of automated reasoning tools in order to make the required reasoning by computer, and show that, in practical circumstances, current automated theorem proving (ATP) technology usually performs very well on this kind of tasks.

Chapter 7 is devoted to a slightly different problem, i.e., that of recognizing problem specifications that can be compiled. *Knowledge compilation* (KC) is one of the techniques that have been proposed in the Artificial Intelligence literature for addressing polynomial intractability of reasoning. The central observation is that, in many reasoning tasks, the input to a (e.g., deduction) problem is split into two parts, with different status: typically the first part is not modified very often (e.g., it is a knowledge base that is not frequently revised), and it is used for several instances of the problem. The idea is to split the deduction problem into two phases: in the first one the fixed part of the input (e.g., the knowledge base) is preprocessed, thus obtaining an appropriate data structure, that is then used to solve the original problem, when the varying part of the input becomes available. The goal of preprocessing is to make the second phase (called *on-line reasoning*) computationally easier (possibly polynomial-time) with respect to reasoning in the case when no preprocessing at all is done, under the requirement that the size of the intermediate data structure is polynomial in the size of the input, in order the storing of the computer information to be practical. Intuitively, the effort spent in the preprocessing phase (*off-line reasoning*) pays off when its computational cost is amortized over the (facilitated) answer to many instances.

We give sufficient conditions for a problem specification (formulated in second-order logic over a finite database) to be compilable, and show how KC reduces to compute a second-order view of the input database, plus a rewriting of the specification using the computed view. Such a rewriting relies on syntactic properties of the specification. We present several examples of compilable queries, both in NP and in higher levels of the Polynomial Hierarchy, from deduction problems to problems on graphs, in order to show how the proposed technique applies.

Chapter 8 is devoted to the description of two highly declarative modelling languages for constraint problems: *NP-Alg* and ConSql, extending relational algebra and sql, respectively. The main motivation under the proposal of such languages is that, in commercially available packages for constraint programming, there is no true integration between the data definition and the constraint modelling and programming languages, with the main disadvantage of providing poor levels of data integrity. Proposed languages, instead, can be actually considered as second-order query languages for relational databases, providing strong integration between constraint modelling and programming and up-to-date technology for storing and querying data. We believe that this feature can permit a wider diffusion of the declarative constraint modelling paradigm in industrial environments, allowing a very strong integration with the information system of the enterprise.

Proceeding with the experience of NP-Spec (15), we provide a formal characterization of the *NP-Alg* language, i.e., of its expressive power and data and combined complexity (in the sense of (128)). Moreover, relying on results from (69), we isolate fragments of the *NP-Alg* language that guarantee that queries that can be expressed in them are polynomially solvable. We also show how well-known polynomial problems can be specified in such fragments. This shows an example of advantages of using a formal language like ESO for problem modelling, i.e., the possibility to apply many theoretical results in the literature of second-order logic to constraint modelling and programming.

Finally, we describe a prototype system that implements the ConSql language by relying on a standard relational DBMS, and exploits local-search techniques for finding solutions. It is worth noting that the implemented local-search solving engine, jLocal, is completely independent on the modelling language, and allows to represent a constraint problem specification by relying on abstract concepts like *search space*, *constraint*, *objective function*, etc. Hence, this can be considered one of the first highly declarative and general-purpose local-search solvers available.

Discussions, description of the limitations of the various techniques, and plans for future research are given at the end of each chapter, while technical issues, like proofs of theorems and details about the examples are delayed to the appendixes.

Chapter 2

Reformulation by safe-delay of constraints

2.1 Introduction

In this chapter, we present a technique for constraint problems reformulation that allows to select constraints in a problem specification that can be ignored in a first step, regardless of the instance. Such constraints can be safely deleted from the specification, hence obtaining a new, simplified, problem, that can be instantiated and solved. In a second step, once a solution of the latter problem has been found, such constraints can be efficiently reinforced, in order to obtain valid solutions of the original specification from arbitrary solutions of the simplified one.

The graph k-coloring problem of Example 1.2.1 offers a simple example of a constraint of this kind. We remind that the problem amounts to find an assignment of nodes to k colors such that:

- Each node has at least one color (*covering*);
- Each node has at most one color (*disjointness*);
- Adjacent nodes have different colors (*good coloring*).

For each instance of the problem, if we obtain a solution neglecting the disjointness constraint, we can always choose for each node one of its colors in an arbitrary way in a later stage (cf. Figure 2.1). We call a constraint with this property a *safe-delay constraint*. It is interesting to note that the standard DIMACS formulation in SAT of k-coloring actually omits the disjointness constraint.

Of course not all constraints are safe-delay: as an example, both the covering and the good coloring constraints are not. Intuitively, identifying the set of constraints of a specification that are safe-delay may lead to several advantages:

- The instantiation phase (cf. Figure 1.1) will typically be faster, since safe-delay constraints are not taken into account. As an example, let's assume we want to use (after instantiation) a SAT solver for the solution of k-coloring on a graph

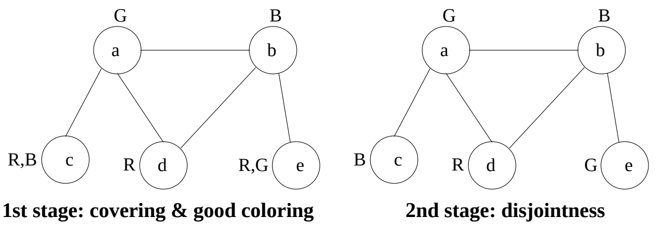

Figure 2.1: Delaying the disjointness constraint in 3-coloring.

with n nodes and e edges. The SAT instance encoding the k-coloring instance – in the obvious way, cf., e.g., (61)– has $n \cdot k$ propositional variables, and a number of clauses which is n, $n \cdot k \cdot (k-1)/2$, and $e \cdot k$ for covering, disjointness, and good coloring, respectively. If we delay disjointness, $n \cdot k \cdot (k-1)/2$ clauses must not be generated.

- Solving the simplified problem, i.e., the one without disjointness, might be easier than the original formulation for some classes of solvers, since removing constraints makes the set of solutions larger. For each instance it holds that:

$$\{\text{solutions of original problem}\} \subseteq \{\text{solutions of simplified problem}\}.$$

  In our experiments using six different solvers, including SAT, integer linear programming, and constraint programming ones, we obtained fairly consistent (in some cases, more than one order of magnitude) speed-ups for hard instances of various problems, e.g., graph coloring and job-shop scheduling. On top of that, we implicitly obtain several good solutions. Results of the experimentation are given in Section 2.4.

- Ad hoc efficient methods for solving delayed constraints may exist. As an example, for k-coloring, the problem of choosing only one color for the nodes with more than one color is $O(n)$.

The architecture we propose is illustrated in Figure 2.2 and can be applied to any system that separates the instance from the specification. It is in some sense similar to the well-known *divide and conquer* technique, cf., e.g., (37), but rather than dividing the instance, we divide the constraints. In general, the first stage will be more computationally expensive than the second one, that, in our proposal, will always be doable in polynomial time.

The goal of this chapter is to understand in which cases a constraint is safe-delay. Our main contribution is the characterization of safe-delay constraints with respect to a semantic criterion on the specification. This allows us to obtain a mechanism for the automated reformulation of a specification that can be applied to a great variety of problems, including the so-called *functional* ones.

The outline of this chapter is as follows. In Section 2.2 we present our reformulation technique, and discuss the adopted methodology in Section 2.3. Afterwards, an experimentation on the effectiveness of the approach is described in Section 2.4, on both benchmark and randomly generated instances, using six different solvers, both

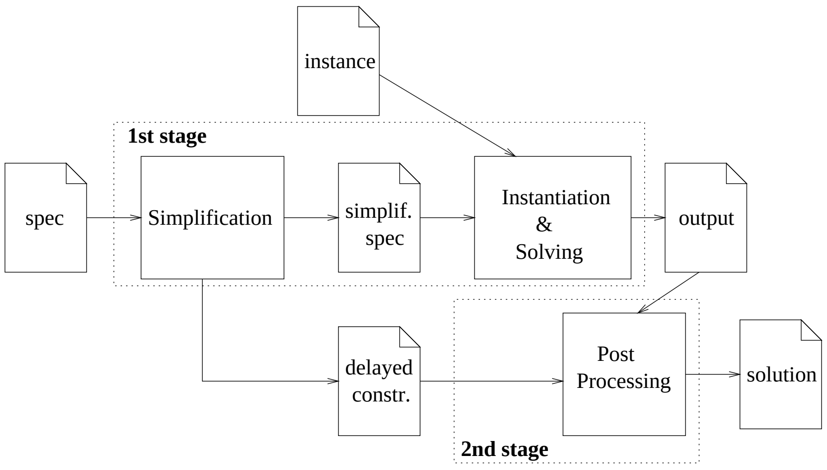

Figure 2.2: Reformulation architecture for safe-delay.

SAT based, linear and constraint programming ones. Finally, discussions, future and related work are presented in Section 2.5.

2.2 Reformulation

In this section we show sufficient conditions for constraints of a specification to be safe-delay. We refer to the architecture of Figure 2.2, with some general assumptions:

Assumption 1: As shown in Figure 2.1, the output of the first stage of computation may –implicitly– contain *several* solutions. As an example, node c can be assigned to either green or blue, and node e to either red or green. In the second stage we do not want to compute all of them, but just to arbitrarily select one. In other words, we focus on search (i.e., satisfaction) problems, with no objective function to be optimized.

Assumption 2: The second stage of computation can only *shrink* the extension of a guessed predicate. Figure 2.3 represents the extensions of the red predicate in the first (R^*) and second (R) stages of Figure 2.1 ($ext(B)$ and $ext(G)$ are unchanged).

This assumption is coherent with the way most algorithms for constraint satisfaction operate: each variable has an associated *finite domain*, from which values are progressively eliminated, until a satisfying assignment is found. Nonetheless, in Subsection 2.2.4 we give examples of problem specifications that are amenable to safe-delay, although with a second stage of different nature.

Identification of safe-delay constraints requires reasoning on the whole specification, taking into account relations between guessed and database predicates. For the sake of simplicity, in Subsection 2.2.1 we will initially focus our attention on *a single monadic* guessed predicate, trying to figure out which constraints concerning it can be delayed. Afterwards, in Subsection 2.2.2 we extend our results to *sets* of monadic guessed predicates, then, in Subsection 2.2.3, to *binary* predicates.

2.2.1 Single monadic predicate

We refer to the 3-coloring specification of Example 1.2.2, reported here for reader's convenience:

$$
\begin{array}{lll}
\exists RGB & \forall X & R(X) \lor G(X) \lor B(X) \ \land \qquad (1.6)\\
& \forall X & R(X) \rightarrow \neg G(X) \ \land \qquad (1.7)\\
& \forall X & R(X) \rightarrow \neg B(X) \ \land \qquad (1.8)\\
& \forall X & B(X) \rightarrow \neg G(X) \ \land \qquad (1.9)\\
& \forall XY & X \neq Y \land R(X) \land R(Y) \rightarrow \neg edge(X,Y) \land \qquad (1.10)\\
& \forall XY & X \neq Y \land G(X) \land G(Y) \rightarrow \neg edge(X,Y) \land \qquad (1.11)\\
& \forall XY & X \neq Y \land B(X) \land B(Y) \rightarrow \neg edge(X,Y), \qquad (1.12)
\end{array}
$$

and focus on one of the guessed predicates, R, trying to find an intuitive explanation for the fact that clauses (1.7–1.8) can be delayed. We immediately note that clauses in the specification can be partitioned into three subsets: NO_R, NEG_R, and POS_R with –respectively– no, only negative, and only positive occurrences of R.

Neither NO_R nor NEG_R clauses can be violated by shrinking the extension of R. Such constraints will be called *safe-forget* for R, because if we decide to process (and satisfy) them in the first stage, they can be safely ignored in the second one (that, by **Assumption 2** above, can only shrink the extension for R). We note that this is just a possibility, and we are not obliged to do that: as an example, clauses (1.7–1.8) will *not* be processed in the first stage.

Although in general POS_R clauses are not safe-forget –because shrinking the extension of R can violate them– we now show that clause (1.6) is safe-forget. In fact, if we equivalently rewrite clauses (1.6) and (1.7–1.8), respectively, as follows:

$$\forall X \quad \neg B(X) \land \neg G(X) \rightarrow R(X) \qquad (1.6')$$

$$\forall X \quad R(X) \rightarrow \neg B(X) \land \neg G(X), \qquad (1.7'–1.8')$$

we note that clause (1.6') sets a lower bound for the extension of R, and clauses (1.7'–1.8') set an upper bound for it; both the lower and the upper bound are $ext(\neg B(X) \land \neg G(X))$. If we use –in the first stage– clauses (1.6,1.9–1.12) for computing $ext(R^*)$ (in place of $ext(R)$), then –in the second stage– we can safely define $ext(R)$ as $ext(R^*) \cap ext(\neg B(X) \land \neg G(X))$, and no constraint will be violated (cf. Figure 2.3). The next theorem shows that is not by chance that the antecedent of (1.6') is semantically related to the consequence of (1.7'–1.8').

Theorem 2.2.1. *Let ψ be an ESO formula of the form:*

$$\exists S_1, \ldots, S_h, S \quad \Xi \quad \land \quad \forall X \ \ \alpha(X) \ \rightarrow \ S(X) \quad \land \quad \forall X \ \ S(X) \ \rightarrow \ \beta(X),$$

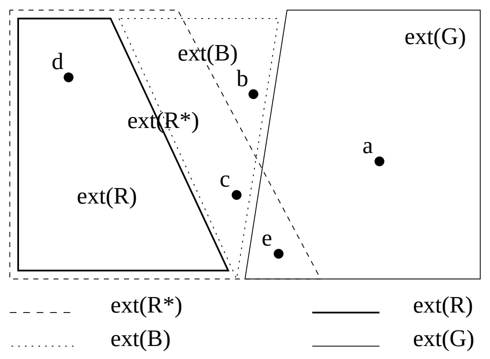

Figure 2.3: Extensions for the 3-coloring specification.

in which S is one of the (all monadic) guessed predicates, Ξ is a conjunction of clauses, both α and β are arbitrary formulae in which S does not occur and X is the only free variable, and such that the following hypotheses hold:

Hyp 1: *S either does not occur or occurs negatively in Ξ;*

Hyp 2: $\models \forall X\ \alpha(X) \rightarrow \beta(X)$.

Let now ψ^s be:

$$\exists S_1, \ldots, S_h, S^* \quad \Xi^* \ \wedge\ \forall X\ \alpha(X) \rightarrow S^*(X),$$

where S^* is a new monadic predicate symbol, and Ξ^* is Ξ with all occurrences of S replaced by S^*, and let ψ^d be:

$$\forall X\ S(X) \leftrightarrow S^*(X) \wedge \beta(X).$$

For every input instance $\mathcal{I}$, and every extension M^* for predicates $(S_1, \ldots, S_h, S^*)$ such that $(M^*, \mathcal{I}) \models \psi^s$, it holds that:

$$((M^* - ext(S^*)) \cup ext(S), \mathcal{I}) \models \psi$$

where $ext(S)$ is the extension of S as defined by M^* and ψ^d.

The proof is delayed to Appendix B. A comment on the relevance of the above theorem is in order. Referring to the architecture of Figure 2.2, ψ is the specification, $\mathcal{I}$ is the instance, ψ^s is the "simplified specification", and $\forall X\ S(X) \rightarrow \beta(X)$ is the "delayed constraint". Solving ψ^s against $\mathcal{I}$ produces –if the instance is satisfiable– a list of extensions M^* (the "output"). Evaluating ψ^d against M^* corresponds to the "PostProcessing" phase in the second stage. The structure of the delayed constraint ψ^d clearly reflects **Assumption 2** above, i.e., that extensions for guessed predicates can only be shrunk in the second stage. Moreover, since the last stage amounts to the evaluation of a first-order formula against a fixed database, it can be done in logarithmic space (cf., e.g., (2)), thus in polynomial time.

In other words, Theorem 2.2.1 says that, for each satisfiable instance $\mathcal{I}$ of the simplified specification ψ^s, each solution M^* of ψ^s can be translated, via ψ^d, to a

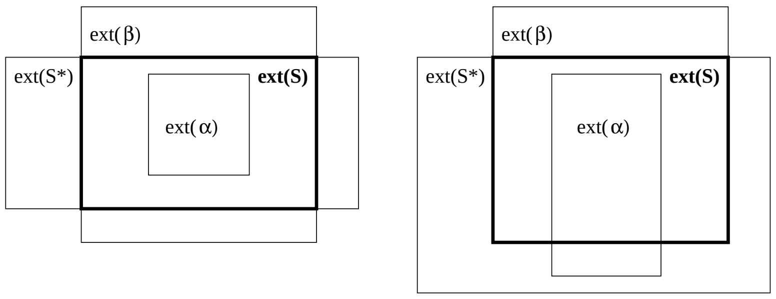

Figure 2.4: Extensions with and without **Hyp 2**.

solution of the original specification ψ; we can also say that $\Xi \wedge \forall X\ \alpha(X) \rightarrow S(X)$ is safe-forget, and $\forall X\ S(X) \rightarrow \beta(X)$ is safe-delay.

Referring to the specification of Example 1.2.2, the distinguished guessed predicate is R, Ξ is the conjunction of clauses (1.9–1.12), and $\alpha(X)$ and $\beta(X)$ are both $\neg B(X) \wedge \neg G(X)$, cf. clauses (1.7'–1.8'). Figure 2.3 represents possible extensions of the red predicate in the first (R^*) and second (R) stages, for the instance of Figure 2.1, and Figure 2.4 (left) gives further evidence that, if **Hyp 2** holds, the constraint $\forall X\ \alpha(X) \rightarrow S(X)$ can never be violated in the second stage.

We are guaranteed that the two-stage process preserves at least one solution of ψ by the following theorem.

Theorem 2.2.2. *Let $\mathcal{I}$, ψ, ψ^s and ψ^d as in Theorem 2.2.1. For every instance $\mathcal{I}$, if ψ is satisfiable, ψ^s and ψ^d are satisfiable.*

To substantiate the reasonableness of the two hypotheses of Theorem 2.2.1, we play the devil's advocate and consider the following example.

Example 2.2.1 (Graph 3-coloring with red self-loops). *In this problem, which is a variation of the one in Example 1.2.2, the input is the same as for Graph 3-coloring, and the question is whether it is possible to find a coloring of the graph with the additional constraint that all self-loops insist on red nodes.*

A specification for this problem can be derived from that of Example 1.2.2 by adding the following constraint:

$$\forall X\ edge(X, X) \rightarrow R(X). \tag{2.1}$$

We immediately notice that now clauses (1.7–1.8) are not safe-delay: intuitively, after the first stage, nodes may be red either because of (1.6) or because of (2.1), and (1.7–1.8) are not enough to set the correct color for a node. Now, if –on top of (1.9–1.12)– Ξ contains also the constraint (2.1), **Hyp 1** is clearly not satisfied. Analogously, if (2.1) is used to build $\alpha(X)$, then $\alpha(X)$ becomes $edge(X, X) \vee (\neg B(X) \wedge \neg G(X))$, and **Hyp 2** is not satisfied. Figure 2.4, right, gives further evidence that the constraint $\forall X\ \alpha(X) \rightarrow S(X)$ can be violated if $ext(S)$ is computed using ψ^d and $ext(\alpha)$ is not a subset of $ext(\beta)$.

Summing up, a constraint with a positive occurrence of the distinguished guessed predicate S can be safely forgotten only if there is a safe-delay constraint that justifies it.

Some further comments about Theorem 2.2.1 are in order. As it can be observed, the theorem proof does not formally require Ξ to be a conjunction of clauses; actually, it can be any formula such that, from any structure M such that $M \models \Xi$, by shrinking $ext(S)$ and keeping everything else fixed we obtain another model of Ξ. As an example, Ξ may contain the conjunct $\exists X\ S(X) \rightarrow \gamma(X)$ (with $\gamma(X)$ a first-order formula in which S does not occur). Secondly, although **Hyp 2** calls for a tautology check – which is not decidable in general– we will see in what follows that many specifications satisfy it *by design*.

2.2.2 Set of monadic predicates

Theorem 2.2.1 states that we can delay some constraints of a specification ψ, by focusing on one of its monadic guessed predicates, hence obtaining a new specification ψ^s, and a set of delayed constraints ψ^d. Of course, the same theorem can be further applied to the specification ψ^s, by focusing on a different guessed predicate, in order to obtain a new simplified specification $(\psi^s)^s$ and new delayed constraints $(\psi^s)^d$. Since, by Theorem 2.2.2, satisfiability of such formulae is preserved, it is afterwards possible to translate, via $(\psi^s)^d$, each solution of $(\psi^s)^s$ to a solution of ψ^s, and then, via ψ^d, to a solution of ψ.

The procedure REFORMULATE in Figure 2.5 deals with the general case of a set of guessed predicates: if the input specification ψ is satisfiable, it returns a simplified specification $\overline{\psi^s}$ and a list of delayed constraints $\overline{\psi^d}$. Algorithm SOLVEBYDELAYING gets any solution of $\overline{\psi^s}$ and translates it, via the evaluation of formulae in the list $\overline{\psi^d}$ –with LIFO policy– to a solution of ψ.

As an example, by evaluating the procedure REFORMULATE on the specification of Example 1.2.2, by focusing on the guessed predicates in the order R, G, B, we obtain as output the following simplified specification $\overline{\psi^s}$, that omits all disjointness constraints (i.e., clauses (1.7–1.9)):

$$\begin{array}{lll}
\exists R^* G^* B & \forall X & R^*(X) \vee G^*(X) \vee B(X) \ \wedge \\
& \forall XY & X \neq Y \wedge R^*(X) \wedge R^*(Y) \rightarrow \neg edge(X,Y) \ \wedge \\
& \forall XY & X \neq Y \wedge G^*(X) \wedge G^*(Y) \rightarrow \neg edge(X,Y) \ \wedge \\
& \forall XY & X \neq Y \wedge B(X) \wedge B(Y) \rightarrow \neg edge(X,Y),
\end{array}$$

and the following list $\overline{\psi^d}$ of delayed constraints:

$$\forall X \quad R(X) \leftrightarrow R^*(X) \wedge \neg G(X) \wedge \neg B(X); \tag{2.2}$$

$$\forall X \quad G(X) \leftrightarrow G^*(X) \wedge \neg B(X). \tag{2.3}$$

It is worth noting that the check that $\forall X\ \beta(X)$ is not a tautology prevents the (useless) delayed constraint $\forall X\ B(X) \leftrightarrow B^*(X)$ to be pushed in $\overline{\psi^d}$.

From any solution of $\overline{\psi^s}$, a solution of ψ is obtained by reconstructing first of all the extension for G by formula (2.3), and then the extension for R by formula (2.2) (synthesized, respectively, in the second and first iteration of the algorithm). Since each delayed constraint is first-order, the whole second stage is doable in logarithmic space (thus in polynomial time) in the size of the instance.

```
Algorithm SolveByDelaying
Input:   a specification $\Phi$, a database $D$;
Output:  a solution of $\langle D, \Phi \rangle$, if satisfiable, 'unsatisfiable' otherwise;

begin
    $\langle \overline{\Phi^s}, \overline{\Phi^d} \rangle$ = Reformulate($\Phi$);
    if ($\langle \overline{\Phi^s}, D \rangle$ is satisfiable) then
      begin
          let $M$ be a solution of $\langle \overline{\Phi^s}, D \rangle$;
          while ($\overline{\Phi^d}$ is not empty) do
          begin
              Constraint $d = \overline{\Phi^d}$.pop();
              $M = M \cup$ solution of $d$; // cf. Theorem 2.2.1
          end;
          return $M$;
      end;
    else return 'unsatisfiable';
end;

Procedure Reformulate
Input:   a specification $\Phi$;
Output:  the pair $\langle \overline{\Phi^s}, \overline{\Phi^d} \rangle$, where $\overline{\Phi^s}$ is a simplified specification, and $\overline{\Phi^d}$
         a stack of delayed constraints;
begin
    Stack $\overline{\Phi^d}$ = the empty stack;
    $\overline{\Phi^s} = \Phi$;
    for each monadic guessed pred. $R$ in $\overline{\Phi^s}$ do
        begin
            partition constraints in $\overline{\Phi^s}$ according to Thm 2.2.1, in:
              $\langle \Xi; \quad \forall X\ \alpha(X) \rightarrow R(X); \quad \forall X\ R(X) \rightarrow \beta(X) \rangle$;
            if the previous step is possible with $\forall X\ \beta(X) \neq$ TRUE then
              begin
                  $\overline{\Phi^d}$.push('$\forall X\ R(X) \leftrightarrow R^*(X) \wedge \beta(X)$');
                  $\overline{\Phi^s} = \Xi^* \wedge \forall X\ \alpha(X) \rightarrow R^*(X)$;
              end;
        end;
    return $\langle \overline{\Phi^s}, \overline{\Phi^d} \rangle$;
end;
```

Figure 2.5: Algorithm for safe-delay in case of a set of monadic predicates.

We also observe that the procedure Reformulate is intrinsically non-deterministic, because of the partition that must be applied to the constraints.

2.2.3 Binary predicates

In this subsection we show how our reformulation technique can be extended in order to deal with specifications with binary (and, in general, n-ary) guessed predicates. This can be formally done by unfolding non-monadic guessed predicates into monadic ones, exploiting the finiteness of the Herbrand domain. In Section 1.2 we already described the unfolding technique and showed how a specification with non-monadic guessed predicates (cf., e.g., Example 1.2.1) can be unfolded into one with only monadic ones (cf. Example 1.2.2). As an example, the output of the unfolding process for the Graph k-coloring specification of Example 1.2.1 for $k = 3$ –up to an appropriate renaming of the three monadic guessed predicates into R, G, B – is exactly the specification of Example 1.2.2.

In Section 1.2, we described the unfolding technique in case the argument to be unfolded can take only a finite and instance independent number of values. In what follows, instead, we make a generalization of the approach, that allows to unfold non-monadic predicates even if domains of all their arguments depend on the input instance. This is because the unfolding technique has to be intended here only as a formal step, and does not have to be performed in practice (cf. forthcoming examples).

The above considerations imply that we can use the architecture of Figure 2.2 for a large class of specifications, including the so called *functional specifications*, i.e., those in which the search space is a (total) function from a finite domain to a finite codomain. A safe-delay functional specification is an ESO formula of the form

$$\exists P \quad \Xi \quad \wedge \quad \forall X\, \exists Y\; P(X,Y) \quad \wedge \quad \forall XYZ \quad P(X,Y) \wedge P(X,Z) \rightarrow Y = Z,$$

where Ξ is a conjunction of clauses in which P either does not occur or occurs negatively. In particular, the disjointness constraints are safe-delay, while the covering and the remaining ones, i.e., Ξ, are safe-forget. Formally, soundness of the architecture on safe-delay functional formulae is guaranteed by Theorem 2.2.1.

Safe-delay functional specifications are quite common; apart from graph coloring, notable examples are Job-shop scheduling and Bin packing, that we consider next.

Example 2.2.2 (Job-shop scheduling (62, Prob. SS18)). *In the Job-shop scheduling problem we have sets (sorts) J for jobs, K for tasks, and P for processors. Jobs are ordered collections of tasks and each task has an integer-valued length (encoded in binary relation L) and the processor that is requested in order to perform it (in binary relation $Proc$). Each processor can perform a task at the time, and tasks belonging to the same job must be performed in their order. Finally, there is a global deadline D that has to be met by all jobs.*

An ESO specification for this problem is as follows. For simplicity, we assume that relation Aft contains all pairs of tasks $\langle k', k'' \rangle$ of the same job such that k' comes after k'' in the given order (i.e., it encodes the transitive closure), and that relation $Time$ encodes all time points until deadline D (thus it contains exactly D tuples). Moreover, we assume that predicate "$\geq$" and function "$+$" are correctly defined on constants in $Time$. It is worth noting that these assumptions do not add any expressive power to the ESO formalism, and can be encoded in ESO with standard techniques.

$$\exists S\ \forall k,t\ S(k,t) \rightarrow K(k) \wedge T(t)\ \wedge \tag{2.4}$$
$$\forall k \exists t\ S(k,t)\ \wedge \tag{2.5}$$
$$\forall k,t',t'' S(k,t') \wedge S(k,t'') \rightarrow t' = t''\ \wedge \tag{2.6}$$
$$\begin{aligned}&\forall k',k'',j,t',t'',l'\ Job(k',j) \wedge Job(k'',j) \wedge \\ &\quad k' \neq k'' \wedge Aft(k'',k') \wedge S(k',t') \wedge S(k'',t'') \wedge \\ &\quad L(k',l') \ \rightarrow\ t'' \geq t' + l'\ \wedge\end{aligned} \tag{2.7}$$
$$\begin{aligned}&\forall k',k'',p,t',t'',l',l'' \\ &\quad Proc(k',p) \wedge Proc(k'',p) \wedge k' \neq k'' \wedge L(k',l') \wedge \\ &\quad L(k'',l'') \wedge S(k',t') \wedge S(k'',t'') \rightarrow \\ &\qquad [(t' \geq t'' \rightarrow t' \geq t'' + l'') \wedge (t' \leq t'' \rightarrow t'' \geq t' + l')] \wedge\end{aligned} \tag{2.8}$$
$$\forall k,t,l\ \ T(k) \wedge S(k,t) \wedge L(k,l) \rightarrow Time(t+l). \tag{2.9}$$

Constraints (2.4–2.6) force a solution to contain a tuple $\langle k,t\rangle$ (t being a time point) for every task k, hence to encode an assignment of exactly a starting time to every task (in particular, (2.6) *assigns at most one starting time to each task). Moreover, constraint* (2.7) *forces tasks that belong to the same job to be executed in their order without overlapping, while* (2.8) *avoids a processor to perform more than one task at each time point. Finally,* (2.9) *forces the scheduling to terminate before deadline D.*

As an example, Figure 2.6(a) and (b) show, respectively, an instance and a possible solution of the Job-shop scheduling problem. The instance consists of 3 jobs (J1, J2, and J3) of, respectively, 4, 3, and 5 tasks each. The order in which tasks belonging to the same job have to be performed is given by the letter in parentheses (a, b, c, d, e). Tasks have to be executed on 3 processors, P1, P2, and P3, that are denoted by different borderlines. Hence, the processor needed to perform a given task is given by the task borderline.

To reformulate the Job-shop scheduling problem, after unfolding the specification in such a way to have one monadic guessed predicate S_t for each time point t, we focus on a time point $\overline{t}$ and partition clauses in the specification in which $S_{\overline{t}}$ does not occur, occurs positively, or negatively, in order to build Ξ, $\alpha(k)$, and $\beta(k)$. The output of this phase is as follows:

- $\alpha(k) \doteq \bigwedge_{t \neq \overline{t}} \neg S_t(k)$ (obtained by unfolding (2.5));
- $\beta(k) \doteq \bigwedge_{t \neq \overline{t}} \neg S_t(k)$ (obtained by unfolding (2.6)).

α and β above clearly satisfy **Hyp 2** of Theorem 2.2.1. Moreover, according to the algorithm in Figure 2.5, by iteratively focusing on all predicates S_t, we can delay all such (unfolded) constraints. It is worth remarking that the unfolding of guessed predicates is needed only to formally characterize the reformulation with respect to Theorem 2.2.1, and must not be performed in practice.

Intuitively, the constraint we delay, i.e. (2.6), forces each task to have at most one start time: thus, by delaying it, we allow a task to have multiple starting times, i.e., the task does not overlap with any other task at any of its start times.

Again, in the second stage, we can arbitrarily choose one of them. We observe that a similar approach has been used in (38) for an optimized ad-hoc translation of this problem into SAT, where propositional variables represent the encoding of *earliest starting times* and *latest ending times* for all tasks, rather than their exact scheduled times, and that it is widely used in current specialized solvers for scheduling problems (cf., e.g., the relevant constructs of OPL that deal with activities and resources). As an example, Figure 2.6(c) shows a solution of the first stage of the reformulated problem for the instance in Figure 2.6(a), which subsumes the solution in Figure 2.6(b).

Example 2.2.3 (Bin packing (62, Prob. SR1)). *In the Bin packing problem (cf. also (91)), we are asked to pack a set I of items, each one having a given size, into a set B of bins, each one having a given capacity. Under the assumption that input instances are given as extensions for relations I, S, B, and C, where I encodes the set of items, B the set of bins, S the size of items (a tuple $\langle i, s\rangle$ for each item i), and C the capacity of bins (a tuple $\langle b, c\rangle$ for each bin b), an ESO specification for this problem is as follows:*

$$\exists P\ \forall i, b\ P(i,b) \rightarrow I(i) \wedge B(b)\ \wedge \tag{2.10}$$

$$\forall i \exists b\ I(i) \rightarrow P(i,b)\ \wedge \tag{2.11}$$

$$\forall i, b, b'\ P(i,b) \wedge P(i,b') \rightarrow b = b' \tag{2.12}$$

$$\forall b, c\ C(b,c) \rightarrow sum\,(\{s \mid P(i,b) \wedge S(i,s)\}) \leq c \tag{2.13}$$

where, to simplify notations, we assume bounded integers to encode the size of items and capacity of bins, and the existence of a function sum that returns the sum of elements that belong to the set given as argument. We remind that bounded integers and arithmetic operations over them do not add expressive power to ESO.

In the above specification, a solution is a total mapping P from items to bins. Constraints force the mapping to be, respectively, over the right relations (2.10)*, total* (2.11)*, monodrome* (2.12)*, and satisfying the capacity constraint for every bin* (2.13).

In particular, by unfolding the guessed predicate P to $|I|$ monadic predicates P_i, one for every item i, and, coherently, the whole specification, the constraints that can be delayed are the unfolding of (2.12), that force an item to be packed in exactly one bin. Thus, by iteratively applying Theorem 2.2.1 by focusing on all unfolded guessed predicates, we intuitively allow an item to be assigned to *several* bins. In the second stage, we can arbitrarily choose one bin to obtain a solution of the original problem. As showed in the previous examples, it is worth noting that arithmetic constraints do not interfere with our reformulation technique. As an instance, in the last example, the "$\leq$" predicate leads to clauses that remain satisfied if the extension of the selected guessed predicate is shrunk, while keeping everything else fixed.

2.2.4 Non-shrink second stages

As specified at the beginning of Section 2.2, in this chapter we have focused on second stages in which the extension of the selected guessed predicate can only be shrunk, while those for the other ones remain fixed.

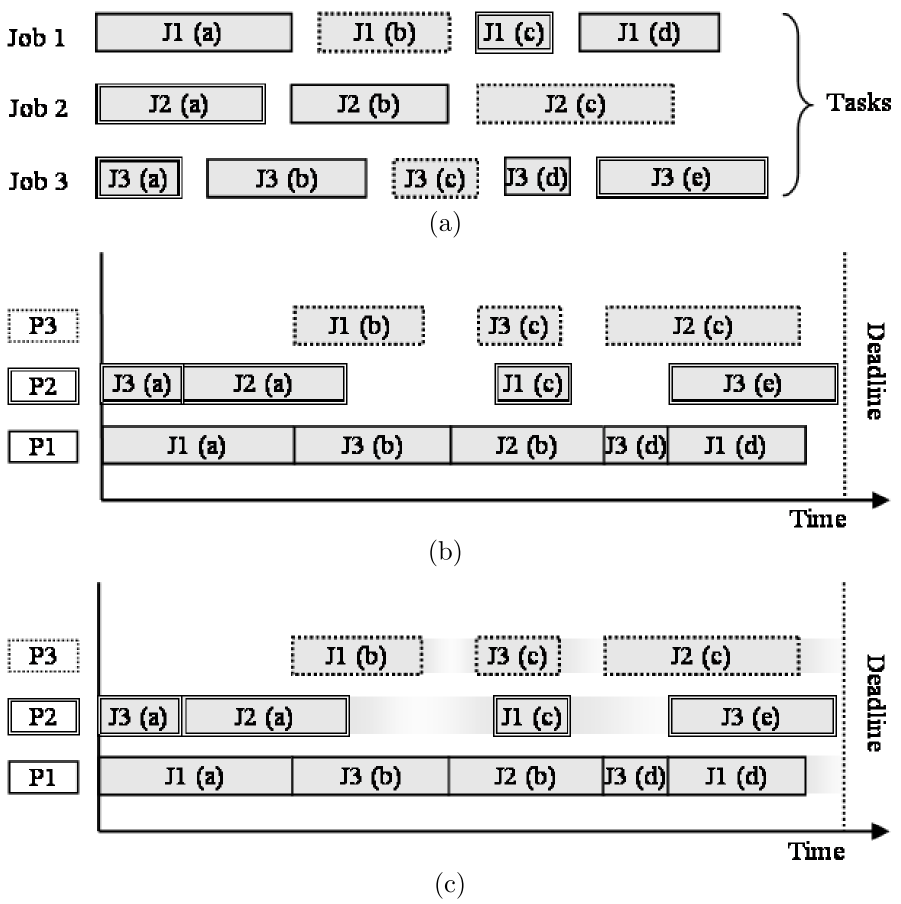

Figure 2.6: (a) An instance of the Job-shop scheduling problem, consisting in 3 jobs of, respectively, 4, 3, and 5 tasks each, to be executed on 3 processors. (b) A possible solution of the whole problem for the instance in (a). (c) A solution of the first stage of the reformulated problem, that subsumes that in (b). Shades indicate multiple good starting times for tasks.

Actually, there are other specifications that are amenable to be reformulated by safe-delay, although with a different kind of second stages. As an example, we show a specification for the Golomb ruler problem.

Example 2.2.4 (Golomb ruler (64, Prob. 3)). *In this problem, we are asked to put m marks $M_1, \ldots, M_m$ on different points on a ruler of length l in such a way that:*

1. *Mark i is put on the left (i.e., before) mark j if and only if $i < j$, and*

2. *The $m(m-1)/2$ distances among pairs of distinct marks are all different.*

By assuming that input instances are given as extensions for unary relations M (encoding the set of marks) and P (encoding the l points on the ruler), and that the function "+" and the predicate "<" are correctly defined on tuples in M and on those in P, a specification for this problem is as follows:

$$\begin{aligned}
\exists G \quad & \forall m, i \; G(m,i) \rightarrow M(m) \wedge P(i) \; \wedge && (2.14)\\
& \forall m \exists i \; M(m) \rightarrow P(m,i) \; \wedge && (2.15)\\
& \forall m, i, i' \; G(m,i) \wedge G(m,i') \rightarrow i = i' \; \wedge && (2.16)\\
& \forall m, m', i, i' \; G(m,i) \wedge G(m',i') \wedge m < m' \rightarrow i < i' \wedge && (2.17)\\
& \forall m, m', i, i', n, n', j, j' \\
& \quad G(m,i) \wedge G(m',i') \wedge G(n,j) \wedge G(n',j') \wedge && (2.18)\\
& \quad m < m' \wedge n < n' \wedge (m < n \vee (m = n \wedge m' < n')) \rightarrow \\
& \qquad (i' - i) \neq (j' - j).
\end{aligned}$$

A solution is thus an extension for the guessed predicate G that is a mapping (2.14–2.16) assigning a point in the ruler to every mark, such that the order of marks is respected (2.17) *and distances between two different marks are all different* (2.18).

Here, the constraint that can be delayed is (2.17), which forces the ascending ordering among marks. By neglecting it, we extend the set of solutions of the original problem with all their permutations. In the second stage, the correct ordering among marks can be enforced in polynomial time.

By unfolding the binary guessed predicate G, we obtain $|M|$ monadic predicates G_m, one for each mark m. Once a solution of the simplified specification has been computed, by focusing on all of them, in order to reinforce the $m(m-1)/2$ unfolded constraints derived from (2.17), we possibly have to *exchange* tuples among pairs of predicates G_m and $G_{m'}$, for all $m \neq m'$, and not to shrink the extensions of single guessed predicates. Hence, Theorem 2.2.1 does not apply. A similar kind of second stage is needed for reformulating some *permutation problems* by safe-delay (cf. Section 3).

Furthermore, a modification of some of the other constraints may be needed to ensure the correctness of the reformulation. As an example, in constraint (2.18) differences must be replaced by their absolute values.

2.3 Safe-delay constraints and other modelling languages

As already claimed in Section 1.2, and as the previous examples show, using ESO for specifying problems wipes out many aspects of state-of-the-art languages which are somehow difficult to take into account (e.g., numbers, arithmetics, constructs for functions, etc.), thus simplifying the task of finding criteria for reformulating problem specifications. However, we already stated that ESO, even if somewhat limited, is not too far away from the modelling languages provided by some commercial systems, and reformulation techniques proposed here can often be straightforwardly applied to specifications written in such languages.

A good example is the AMPL specification of the Graph k-coloring problem shown in Figure 1.3. In this case, the reformulated specification can be obtained by replacing the "CoveringAndDisjointness" constraint with the following one:

```
s.t. Covering {x in NODES}
     sum {c in COLORS} Coloring[x,c] >= 1;
```

thus leading exactly to the reformulated specification of Example 1.2.2.

As for languages that admit non linear constraints, e.g., OPL, it is possible to write a different specification using integer variables for the colors and inequality of colors between adjacent nodes. In this case it is not possible to separate the disjointness constraint from the other ones, since it is implicit in the definition of the domains. Of course, study of safe-delay constraints is relevant also for such languages, because we can always specify in OPL problems such as the one of Example 2.2.4, that has such constraints.

We also note that, as shown in Examples 2.2.2, 2.2.3, and 2.2.4, we can consider useful syntactic sugar for encoding bounded integers, operations and relations such as "*sum*", "+", "$\leq$", etc., without adding expressive power to ESO.

For what concerns the experimentation, it must be observed that a specification written in ESO naturally leads to a translation into a SAT instance. For this reason, we have chosen to use, among others, SAT solvers for the experimentation of the proposed technique. Moreover we considered also the impressive improving in performances recently shown by state-of-the art SAT solvers.

As already claimed in Section 1, the effectiveness of a reformulation technique is expected to strongly depend on the particular solver used. To this end, we solved the same set of instances with SAT solvers of very different nature (cf. Section 5.5). Finally, since it is known that state-of-the-art linear and constraint programming systems may perform better than SAT on some problems, we repeated the experimentation by using commercial systems CPLEX (linear) and SOLVER (non-linear), cf. `www.ilog.com`.

2.4 Experimental results

We made an experimentation of our reformulation technique on 3-coloring (randomly generated instances), k-coloring (benchmark instances from the DIMACS repository (46), and job-shop scheduling (benchmark instances from OR library (104)), using both SAT-based solvers (and an ad hoc program (26) for the instantiation stage), and the constraint and linear programming system OPL (124), obviously using it as a pure modelling language, and omitting search procedures.

As for the SAT-based experimentation, we used four solvers of very different nature: the DPLL-based complete systems ZCHAFF (100) and SATZ (87), and the local-search based incomplete solvers WALKSAT (115) and BG-WALKSAT (133) (the last one being guided by "backbones"). We solved all instances both with and without delaying constraints. As for OPL, we wrote both a linear and a non-linear specification for the above problems, and applied our reformulation technique to the linear one (cf. Section 2.3).

Experiments were executed on an Intel 2.4 GHz Xeon bi-processor computer. The size of instances was chosen so that our machine is able to solve (most of) them in more than few seconds, and less than one hour. In this way, post-processing time, i.e., the time for evaluating delayed constraints, is negligible with respect to solving time, and comparison can be done only on the latter.

In what follows, we refer to the *saving percentage*, defined as the ratio:

$$(\textit{time\_with\_disj.} - \textit{time\_without\_disj.})\,/\,\textit{time\_with\_disj.}$$

Graph 3-coloring. We solved the problem on 1500 randomly generated graph instances with 500 nodes each. The number of edges varies, and covers the phase transition region (32): the ratio (# of directed edges/# of nodes) varies between 2.0 and 6.0. In particular, we considered sets of 100 instances for each fixed number of edges, and solved each set both with and without delaying disjointness constraints (timeout was set at 1 hour). Table 2.1 shows overall solving times for each set of instances, for all the SAT solvers under consideration.

As it can be observed, the saving percentage highly depends both on the edges/nodes ratio, and on the solver. In particular, ZCHAFF seems not to be positively affected by safe-delay, and, for some classes of instances, e.g., those with 1500 edges, it seems to be negatively affected (-28.40%). On the other hand, both local-search based SAT solvers, i.e., WALKSAT and BG-WALKSAT show a consistent improvement with our reformulation technique, saving from 13% to 17% for hard instances. Even if they are incomplete solvers, they have been always able to find a solution for satisfiable instances, except for the class of input graphs with 1100 edges: in this case, they found a solution on the 40% (WALKSAT) and 50% (BG-WALKSAT) of positive instances. Delaying disjointness constraints does not alter this percentage in a significant way. Also SATZ benefits from safe-delay, with savings from 22% to 26% for hard instances, even if underconstrained instances (e.g., those with 700 edges) highlight poorer performances (-49.83%). It is interesting to note that the best technology, i.e., local search, is improved.

For what concerns CPLEX instead, experimental results do not highlight saving in performances, since, in many cases, for the same set of 1500 instances, either both solving times were negligible, or a timeout occurred both with and without disjointness constraints.

Graph k-coloring. We solved the k-coloring problem on several benchmark instances, with k close to the optimum. Results of our SAT-based experiments are

Table 2.1: Experimental results for 3-coloring on random instances with 500 nodes (100 instances for each fixed number of edges). Solving times (in seconds, with timeout of 1 hour) are relative to each set of 100 instances.

Und.		ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
edges	e/n	$\sum$ w/ disj.	$\sum$ w/o disj.	% sav.	$\sum$ w/ disj.	$\sum$ w/o disj.	% sav.	$\sum$ w/ disj.	$\sum$ w/o disj.	% sav.	$\sum$ w/ disj.	$\sum$ w/o disj.	% sav.
500	2.0	0.26	0.26	0.00%	0.92	0.60	34.55%	2.43	1.85	23.97%	10.17	7.17	29.50%
600	2.4	0.13	0.22	-69.23%	2.22	1.82	18.05%	3.63	3.00	17.43%	8.97	5.08	43.37%
700	2.8	0.22	0.12	45.45%	8.48	7.58	10.61%	9.92	8.60	13.28%	7210.96	10804.02	-49.83%
800	3.2	0.2	0.1	50.00%	9.33	8.58	8.04%	10.95	9.82	10.35%	7208.19	7240.53	-0.45%
900	3.6	0.18	0.14	22.22%	18.18	16.52	9.17%	19.88	17.87	10.14%	14408.3	14403.53	0.03%
1000	4.0	0.16	0.07	56.25%	48.77	42.82	12.20%	48.62	42.97	11.62%	18787.4	21603.45	-14.99%
1100	4.4	11.25	12.07	-7.29%	119.70	105.50	11.86%	119.47	105.47	11.72%	11825.44	8772.81	25.81%
1200	4.8	80537.81	86843.41	-7.83%	129.35	112.05	13.37%	129.75	113.78	12.31%	16686.21	12279.19	26.41%
1300	5.2	1497.69	1441.24	3.77%	134.50	115.80	13.90%	138.37	117.58	15.02%	700.14	544.35	22.25%
1400	5.6	60.4	59.77	1.04%	145.32	119.93	17.47%	145.53	124.68	14.33%	95.09	78.27	17.69%
1500	6.0	18.77	24.1	-28.40%	147.68	125.25	15.19%	152.82	128.97	15.61%	31.56	26.95	14.61%

shown in Table 2.2. As it can be observed, also here the effectiveness of the reformulation technique varies among the different solvers. In particular, ZCHAFF benefits by safe-delay on several instances, both positive and negative, but not on all of them. On the other hand, for local search solvers WALKSAT and BG-WALKSAT, delaying disjointness constraints always (except for very few cases) speeds-up the computation (usually by 20-30%). The same happens when using SATZ with even higher savings, even if this solver timeouts for several instances.

As for OPL instead, we have mixed evidence, as Table 2.3 shows. In particular, it is not the case that the linear specification (solved using CPLEX) is always more efficient than the non-linear one (solved using SOLVER), or vice versa: actually, the ratio between the solving time of CPLEX and SOLVER is highly variable. However, by focusing on the class of instances for which the linear specification is more efficient than the non-linear one, delaying the disjointness constraint further improves performances.

It is worth noting that Tables 2.2 and 2.3 show solving times for only those instances for which at least one solver was able to terminate in less than 1 hour. For this reason, even if we used the same initial set of instances for the various solvers (SAT, CPLEX, and SOLVER), instances in the two tables do not coincide.

Table 2.2: Solving times (seconds) for k-coloring (SAT solvers).

			ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
Instance	Colors	Solvable?	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.
anna	10	N	24.87	15.02	39.61	6.48	5.71	11.83	6.2	4.0	4.57	–	–	–
anna	11	Y	0.01	0.01	0.00	0.08	0.05	40.00	0.1	0.06	33.33	0.29	0.09	68.97
david	10	N	15.15	13.04	13.93	5.58	4.78	14.33	5.52	4.66	15.41	–	–	–
david	11	Y	0.1	0.1	0.00	0.07	0.05	25.00	0.07	0.05	25.00	0.63	0.08	87.30
DSJC125.5	8	N	41.42	4.04	90.25	mem	mem	–	mem	mem	–	–	–	–
DSJC250.5	10	N	–	52.69	>98.54	mem	mem	–	mem	mem	–	–	–	–
DSJC500.1	5	N	11.14	1.55	86.09	mem	mem	–	mem	mem	–	158.67	145.33	8.41
le450_5a	5	Y	17.23	0.91	94.72	5.42	4.91	9.23	5.51	4.97*	9.97	96.83	77.16	20.31
le450_5b	5	Y	27.77	1.36	95.10	6.17*	5.10	17.30	6.15*	5.48	10.84	104.62	93.33	10.79
le450_5c	5	Y	0.02	0.02	0.00	10.00*	8.67	13.33	9.65	9.23*	4.32	22.45	20.02	10.82
le450_5c	9	Y	11.20	1.81	83.84	1.20	1.20	0.00	1.47	1.35	7.95	–	–	–
le450_5d	5	Y	0.09	1.20	-1233.33	8.52	8.15	4.31	8.42	8.40	0.20	6.00	5.20	13.33
miles500	9	N	80.19	54.55	31.97	9.08	8.51	6.24	9.70	7.38	23.92	–	–	–
miles500	20	Y	0.01	0.01	0.00	0.53	0.36	31.25	0.63	0.46	26.32	8.36	3.76	55.02
mulsol.i.2	30	N	–	–	–	28.83	22.85	20.75	30.18	24.16	19.93	–	–	–
mulsol.i.2	31	Y	0.01	0.01	0.00	4.25	2.36	44.31	4.21	2.80	33.60	5.10	2.67	47.65
mulsol.i.3	30	N	–	–	–	29.20	23.03	21.12	31.47	24.38	22.51	–	–	–
mulsol.i.3	31	Y	0.01	0.01	0.00	4.22	2.48	41.11	4.60	3.05	33.70	5.04	2.70	46.43
mulsol.i.4	30	N	–	–	–	29.40	23.58	19.78	30.93	24.81	19.77	–	–	–
mulsol.i.4	31	Y	0.01	0.01	0.00	4.03	2.38	40.91	4.98	2.76	44.48	5.08	2.73	46.26
mulsol.i.5	30	N	–	–	–	26.70	21.38	19.91	28.48	22.63	20.54	–	–	–
mulsol.i.5	31	Y	0.01	0.01	0.00	4.63	2.53	45.32	5.42	2.88	46.77	5.14	2.74	46.69
myciel5	5	N	413.99	1714.26	<-314.08	1.33	1.16	12.50	1.33	1.15	13.75	52.89	39.25	25.79
myciel5	6	Y	0.01	0.01	0.00	0.02	0.02	0.00	0.01	0.01	0.00	0.10	0.04	60.00
queen8_8	9	Y	1397.23	2144.76	-53.50	5.23	3.95	24.52	5.25	4.13	21.27	0.27	0.22	18.52
queen9_9	10	Y	–	–	–	8.22*	7.31*	10.95	8.43*	7.61*	9.68	56.91	45.51	20.03
queen11_11	13	Y	553.46	863.14	-55.95	15.13	9.71	35.79	14.47	11.63	19.59	205.66	175.62	14.61
queen14_14	17	Y	17.75	114.14	-543.04	8.27	11.82	-42.94	7.40	9.08	-22.75	943.30	708.92	24.86
queen8_12	12	Y	0.13	1.78	-1269.23	5.27	4.68	11.08	6.03	5.71	5.25	0.35	0.25	28.57

('–' means that the solver did not terminate in one hour, while 'mem' that an out-of-memory error occurred. A '*' means that the local search solver did not find a solution).

Table 2.3: Solving times (seconds) for k-coloring (OPL).

			CPLEX			SOLVER
Instance	Colors	Solvable?	w/ disj.	w/o disj.	% sav.	w/ disj.
anna	10	N	0.76	0.62	18.42	–
anna	11	Y	0.64	0.61	4.69	0.87
david	10	N	0.23	0.25	-8.70	–
david	11	Y	0.56	0.64	-14.29	0.92
DSJC125.1	4	N	–	–	–	0.48
DSJC125.1	5	Y	–	–	–	4.77
DSJC125.5	8	N	15.32	12.45	18.73	10.81
DSJC125.5	20	Y	–	–	–	9.68
DSJC125.5	25	Y	–	–	–	0.59
DSJC125.9	21	N	1408.23	937.69	33.41	–
DSJC250.1	9	Y	–	–	–	2.61
DSJC250.5	10	N	–	3160.31	>12.21	–
DSJC500.1	11	N	2.53	1.65	34.78	–
DSJC500.1	12	Y	19.22	18.15	5.57	–
fpsol2.i.2	21	N	22.79	14.24	37.52	–
fpsol2.i.2	31	Y	–	–	–	0.34
fpsol2.i.2	32	Y	3305.41	–	<-8.91	1.25
fpsol2.i.3	31	Y	–	–	–	0.64
fpsol2.i.3	32	Y	1572.75	–	<-128.90	0.51
games120	8	N	0.97	1.05	-8.25	–
games120	9	Y	0.97	0.88	9.28	0.59
huck	10	N	0.57	0.41	28.07	–
huck	11	Y	0.72	0.8	-11.11	0.73
jean	9	N	0.35	0.43	-22.86	–
jean	10	Y	0.24	0.22	8.33	1.78
le450_15a	13	N	9.51	6.44	32.28	–
le450_15a	16	Y	–	–	–	3.43
le450_15b	16	Y	–	–	–	576.26
le450_25a	21	N	84.73	17.32	79.56	–
le450_25a	25	Y	3536.41	–	<-1.80	1.48
le450_25b	25	Y	–	–	–	0.89
le450_5a	4	N	2.24	2.11	5.80	0.76
le450_5a	5	Y	–	–	–	3.68
le450_5b	4	N	2.63	2.47	6.08	0.81
le450_5b	5	Y	–	–	–	1.67
le450_5c	4	N	5.64	6.05	-7.27	0.60
le450_5c	5	Y	690.33	2408.61	-248.91	1.51
le450_5d	4	N	5.89	6.03	-2.38	0.75
le450_5d	5	Y	549.78	651.46	-18.49	0.93
miles1000	42	Y	–	–	–	0.97
miles250	7	N	0.67	0.53	20.90	–
miles250	8	Y	1.21	0.98	19.01	0.86
miles500	19	N	2.31	1.54	33.33	–
miles500	20	Y	2.15	2.01	6.51	0.81
miles750	31	Y	134.88	194.24	-44.01	0.78
mulsol.i.1	49	Y	–	92.13	>97.44	1.68
mulsol.i.2	31	Y	26.75	48.24	-80.34	0.66
mulsol.i.3	31	Y	55.77	61.13	-9.61	0.91
mulsol.i.4	31	Y	47.46	53.52	-12.77	1.57
mulsol.i.5	31	Y	166.85	–	<-2057.63	0.49
myciel3	3	N	0.86	0.79	8.14	0.74
myciel3	4	Y	1.25	0.98	21.60	0.81
myciel4	4	N	5.22	4.78	8.43	0.79
myciel4	5	Y	1.57	0.88	43.95	0.68
myciel5	5	N	–	–	–	1139.41

('–' means that the solver did not terminate in one hour.)

Table 2.3: (continued)

			CPLEX			SOLVER
Instance	Colors	Solvable?	w/ disj.	w/o disj.	% sav.	w/ disj.
myciel5	6	Y	1.77	0.87	50.85	0.73
myciel6	7	Y	1.07	0.91	14.95	0.65
myciel7	8	Y	1.42	1.33	6.34	0.58
queen10_10	12	Y	93.86	44.57	52.51	0.73
queen10_10	15	Y	3.24	3.26	-0.62	1.59
queen11_11	13	Y	–	125.1	>96.53	39.44
queen12_12	14	Y	–	–	–	395.71
queen12_12	15	Y	464.78	228.41	50.86	8.45
queen13_13	16	Y	355.31	1055.29	-197.01	168.83
queen14_14	17	Y	–	2220.31	>38.32	1.57
queen16_16	19	Y	–	–	–	425.75
queen5_5	4	N	0.81	1.02	-25.93	0.61
queen5_5	5	Y	0.76	0.65	14.47	0.58
queen6_6	6	N	5.62	5.71	-1.60	1.79
queen6_6	7	Y	10.84	6.05	44.19	1.35
queen7_7	6	N	0.99	1.13	-14.14	0.80
queen7_7	7	Y	1.89	3.07	-62.43	0.69
queen8_12	11	N	1.51	1.47	2.65	1552.69
queen8_12	12	Y	75.85	221.24	-191.68	1.29
queen8_8	9	Y	–	2411.25	>33.02	1.65
queen9_9	10	Y	–	–	–	126.41
zeroin.i.1	49	Y	–	–	–	0.77
zeroin.i.2	30	Y	48.86	68.36	-39.91	1.81
zeroin.i.3	30	Y	–	56.26	>98.44	1.39

('–' means that the solver did not terminate in one hour.)

Job shop scheduling. We considered 40 benchmark instances known as LA01, ..., LA40, with the number of tasks ranging from 50 to 225, number of jobs from 10 to 15, and number of processors ranging from 5 to 15. However, in order to make our solvers (especially the SAT ones) able to deal with such large instances, we reduced (and rounded) all task lengths and the global deadline by a factor of 20 (original lengths were up 100). In this way, we obtained instances that are good approximations of the original ones, but with much smaller time horizons, hence less propositional variables must be generated.

SAT solving times are listed in Table 2.4 for different values for the deadline. Again, we have a mixed evidence for what concerns ZCHAFF, which benefits from safe-delay on many but not all instances, while savings in time are always positive when using local search solvers WALKSAT and BG-WALKSAT (even when they are not able to find a solution for positive instances, delaying constraints makes them terminate earlier). As for SATZ, savings in performances are often very high, even if this solver is able to solve only a small portion of the instance set. Interestingly, the blow-down in the number of clauses due to safe-delay, in some cases makes the solver able to handle some large instances (cf. Table 2.4, e.g., instance LA06 with deadline 50), preventing the system from going out of memory (even if, in many cases, the out-of-memory error changes to a timeout).

For what concerns the experiments in CPLEX instead, this solver seems not to be much affected by delaying constraints (or even affected negatively) on this problem, and anyway it is slower than SOLVER. For this reason, detailed results are omitted. We also remind that OPL, when used with non-linear specifications, has specialized constructs for dealing with scheduling problems, that propagate very efficiently. This feature makes impossible a fair experimentation of this reformulation technique with SOLVER, since the relevant specification does not explicitly make use of variables for tasks' starting times.

Summing up, we solved several thousands of instances. Delaying of constraints seems to be useful when using a SAT solver. In particular, local search solvers like WALKSAT and BG-WALKSAT almost always benefit from the reformulation. The same happens when using SATZ. As for ZCHAFF, we have mixed evidence, since in some cases it seems not to be affected by safe-delay (cf., e.g., results in Table 2.1), or affected negatively, while in others it benefits from the reformulation (cf., e.g., Tables 2.2 and 2.4).

As far as CPLEX is concerned, we have mixed evidence. First of all, as already observed, it is not always the case that CPLEX is always faster than SOLVER, or vice versa. However, in those cases in which the linear specification is faster, it is possible to find several instances for which delaying constraints speeds-up the computation.

As a concluding remark, it is interesting to observe that the experimentation shows that very often delaying constraints improves the performances of the best solver for each instance. Performances of other –not optimal– solvers instead can be worse.

Table 2.4: Solving times (seconds) for job shop scheduling.

						ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
Instance	Proc	Tasks	Jobs	Deadline	Solvable?	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.
la01	5	50	10	33	N	0.69	0.85	-23.19	39.83	31.13	21.84	39.55	31.21	21.07	–	–	–
la01	5	50	10	34	Y	0.13	1.24	-853.85	40.77*	31.42*	22.94	40.72*	31.95*	21.53	154.49	1.76	98.86
la02	5	50	10	32	N	1.75	1.61	8.00	36.63	28.63	21.84	37.95	29.92	21.17	–	–	–
la02	5	50	10	33	Y	2.34	1.07	54.27	37.37*	28.80*	22.93	38.23*	29.10*	23.89	–	–	–
la03	5	50	10	31	N	2.77	2.10	24.19	32.78	25.37	22.62	32.92	25.11	23.70	–	–	–
la03	5	50	10	32	Y	1.92	0.31	83.85	32.85*	25.53*	22.27	32.67*	26.31*	19.44	–	–	–
la04	5	50	10	29	N	1.11	1.06	4.50	33.22	26.10	21.42	33.48	26.06	22.15	904.38	189.44	79.05
la04	5	50	10	30	Y	4.05	2.83	30.12	33.25*	26.22*	21.15	34.47*	26.02*	24.52	699.77	295.12	57.83
la05	5	50	10	28	N	1.33	0.99	25.56	28.92	22.83	21.04	29.25	23.10	21.03	–	–	–
la05	5	50	10	29	Y	0.29	0.57	-96.55	26.42	18.40	30.35	27.73*	19.46	29.81	0.89	0.58	34.83
la06	5	75	15	46	N	–	–	–	78.10	62.92	19.44	79.98	63.00	21.23	mem	–	–
la06	5	75	15	50	Y	0.44	1.89	-329.55	77.32	60.20*	22.14	78.77*	60.37*	23.36	mem	4.87	100.00
la07	5	75	15	43	N	–	–	–	78.35	62.21	20.59	80.10	62.55	21.91	–	–	–
la07	5	75	15	50	Y	1.27	0.98	22.83	78.62*	60.78*	22.68	78.08*	59.40*	23.93	mem	–	–
la08	5	75	15	42	N	–	–	–	77.02	60.72	21.16	76.18	60.17	21.02	–	–	–
la08	5	75	15	43	Y	262.44	181.53	30.83	76.83*	62.02*	19.28	77.25*	61.07*	20.95	–	–	–
la09	5	75	15	46	N	–	–	–	86.58	67.20	22.39	86.57	67.15	22.43	mem	–	–
la09	5	75	15	50	Y	0.31	5.76		85.73*	65.40*	23.72	83.65*	63.30	24.33	mem	3108.10	100.00
la10	5	75	15	46	N	–	–	–	80.80	64.86	19.72	81.42	63.07	22.54	mem	–	–
la10	5	75	15	50	Y	2.23	15.44		80.10*	61.90*	22.72	80.97*	61.05*	24.60	mem	–	–
la16	10	100	10	47	N	155.26	90.71	41.58	69.33	51.73	25.38	67.47	51.11	24.23	mem	–	–
la16	10	100	10	48	Y	151.64	18.71	87.66	69.87*	51.55*	26.22	70.70*	51.53*	27.11	mem	–	–
la17	10	100	10	39	N	5.42	3.72	31.37	57.03	44.46	22.03	56.23	43.78	22.14	–	–	–
la17	10	100	10	40	Y	3.68	4.19	-13.86	57.82*	44.20*	23.55	58.80*	43.71*	25.65	–	–	–
la18	10	100	10	41	N	6.31	4.21	33.28	62.30	47.53	23.70	61.12	45.48	25.58	–	–	–
la18	10	100	10	42	Y	5.75	3.27	43.13	63.72*	47.50*	25.45	62.17*	46.85*	24.64	–	–	–
la19	10	100	10	42	N	50.04	33.09	33.87	63.28	47.17	25.47	63.97	47.4	25.90	–	–	–
la19	10	100	10	43	Y	120.68	154.67	-28.17	64.83*	48.53*	25.14	63.52*	47.58*	25.09	–	3524.02	>2.11
la20	10	100	10	44	N	6.04	5.33	11.75	66.52	50.68	23.80	66.22	49.53	25.20	mem	–	–
la20	10	100	10	45	Y	20.86	13.91	33.32	68.87*	50.60*	26.52	67.47*	49.40*	26.78	mem	–	–
la22	10	150	15	48	N	–	–	–	mem	mem	–	mem	mem	–	mem	mem	–
la22	10	150	15	50	Y	1173.04	619.5	47.19	mem	mem	–	mem	mem	–	mem	mem	–
la23	10	150	15	50	N	–	934.79	>48.07	mem	mem	–	mem	mem	–	mem	mem	–
la23	10	150	15	53	Y	–	1028.56	42.86	mem	mem	–	mem	mem	–	mem	mem	–
la24	10	150	15	46	N	138.15	123.37	10.70	mem	mem	–	mem	mem	–	mem	mem	–
la24	10	150	15	48	Y	–	1083.31	>39.82	mem	mem	–	mem	mem	–	mem	mem	–
la25	10	150	15	48	N	356.56	344.02	3.52	mem	mem	–	mem	mem	–	mem	mem	–
la25	10	150	15	50	Y	438.27	510.21	-16.41	mem	mem	–	mem	mem	–	mem	mem	–
la29	10	200	20	50	N	705.09	216.74	69.26	mem	mem	–	mem	mem	–	mem	mem	–
la29	10	200	20	75	Y	–	1387.73	>22.90	mem	mem	–	mem	mem	–	mem	mem	–

(‘–’ means that the solver did not terminate in one hour, while ‘mem’ that an out-of-memory error occurred. A ‘*’ means that the local search solver did not find a solution).

Table 2.4: (continued)

						ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
Instance	Proc	Tasks	Jobs	Deadline	Solvable?	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.	w/ disj.	w/o disj.	% sav.
la36	15	225	15	62	N	1238.04	782	36.84	mem	mem	–	mem	mem	–	mem	mem	–
la36	15	225	15	65	Y	–	527.86	>70.67	mem	mem	–	mem	mem	–	mem	mem	–
la38	15	225	15	58	N	–	844.58	>53.08	mem	mem	–	mem	mem	–	mem	mem	–
la38	15	225	15	75	Y	1329.74	–	<-35.36	mem	mem	–	mem	mem	–	mem	mem	–
la39	15	225	15	61	N	1193.76	1773.00	-48.52	mem	mem	–	mem	mem	–	mem	mem	–
la39	15	225	15	62	Y	787.54	918.24	-16.60	mem	mem	–	mem	mem	–	mem	mem	–
la40	15	225	15	59	N	989.98	712.96	27.98	mem	mem	–	mem	mem	–	mem	mem	–
la40	15	225	15	75	Y	1154.18	1256.54	-8.87	mem	mem	–	mem	mem	–	mem	mem	–

('–' means that the solver did not terminate in one hour, while 'mem' that an out-of-memory error occurred. A '*' means that the local search solver did not find a solution).

2.5 Discussion and future research directions

In this chapter we have shown a simple reformulation architecture and proven its soundness for a large class of problems. The reformulation allows to delay the solution of some constraints, often resulting in faster solving. In this way, we have shown that reasoning on the specification can be very effective, at least on some classes of solvers.

In many of the proposed examples, the constraints that have been delayed are those stating that a variable must be assigned to *at most one* domain value (cf., e.g., the disjointness constraints in Graph k-coloring). It may be wondered that this fact is somehow implicit in the CP paradigm, where variables are always assigned to *exactly one* value. However, in this thesis we take a Knowledge Representation approach to problem modelling and reformulation, aiming to have a strong decoupling of the problem specification from the solving technology used. Hence, as already mentioned in Section 1.3, the usefulness of this and other reformulation techniques must be evaluated in relation to the class of solvers used. As for the constraints delayed in the proposed examples, we observe that they need to be explicitly stated when using Mathematical Programming (cf., e.g., the AMPL specification of Graph k-coloring in Figure 1.3), or SAT technology, and, as experimental results show, delaying them often greatly improves performances of such solvers.

Although Theorem 2.2.1 calls for a tautology checking (cf. **Hyp 2**), we have shown different specifications for which this test is immediate. Furthermore, we believe that, in practice, an automated theorem prover (ATP) can be used to reason on specifications, thus making it possible to automatically perform the task of choosing constraints to delay. Actually, in Chapter 6 we show that state-of-the-art ATPs usually perform very well in similar tasks (i.e., detecting and breaking symmetries and detecting functional dependencies on problem specifications).

The proposed technique focuses on a form of reformulation that partitions the first-order part of a specification. This basic idea can be generalized, as an example by evaluating in both stages of the computation a constraint (e.g., (2.1)), or to allow non-shrink second stages (cf., e.g., the specification for the Golomb ruler problem in Example 2.2.4), in order to allow reformulation for a larger class of specifications (an extension of the technique to a class of permutation problem is studied in Chapter 3). Even more generally, the second stage may amount to the evaluation of a second-order formula. In the future, we plan –with a more extensive experimentation– to check whether such generalizations are effective in practice.

Another important issue is to understand the relationships between delaying constraints and other techniques, e.g., symmetry breaking (cf. Chapter 4). In fact, it is not always the case that delaying constraints, and so making the set of solutions larger, improves the solving process. Adding, e.g., symmetry-breaking constraints is a well known technique that may reach the same goal with the opposite strategy, i.e., reducing the set of solutions. Currently, it is not completely clear in which cases removing constraints results in better performances with respect to adding more constraints to the specification itself, even if it seems that an important role is played by the nature of constraints we remove or add, e.g., by their amenability to propagation in the search tree, together with the nature of the solver used, e.g., backtracking, linear, or based on local search. As for the latter class of solvers, it is known that enlarging the set of solutions can significantly speed-up performances (cf., e.g., (106)). Our experiments on WALKSAT and BG-WALKSAT confirm this thesis.

Finally, it is our goal to rephrase the theoretical results into rules for automatically reformulating problem specifications given in more complex languages, e.g. AMPL and OPL, which have higher-level built-in constructs.

Results in this chapter have been published in (20).

Chapter 3

Safe-delay constraints for permutation problems

3.1 Introduction

In this chapter, we extend the reformulation technique by safe-delay proposed in Chapter 2 to the context of *permutation problems*. As already argued in Subsection 2.2.4, some problem specifications may exhibit constraints that can be safely delayed, even if with a second stage of different nature with respect to that described in Chapter 2. This is the case of permutation problems. In this context, the scenario is more complex, in that, to allow some of the constraints to be delayed, a slight modification of the other constraints is needed, to ensure correctness (cf. Section 3.3). The architecture we propose is yet the one depicted in Figure 2.2, even if the "Simplification" stage is more complex.

In particular, this chapter focuses on the subclass of permutation problems characterized by constraints that bind an element of the permutation to the next or previous one only. Problems in this class arise frequently both in theory and in practice: NP-complete *Hamiltonian path*, *Permutation flow-shop scheduling*, and the *Tiling* problems are good examples.

The outline is as follows: in Section 3.2 we show how permutations can be formulated in ESO and provide the specification of a prototypical problem in the class we focus on: Hamiltonian path (HP), which we carry on as running example. In Section 3.3 we show a first reformulation of the HP problem, while in Section 3.4 we provide further examples. The reformulation technique presented in Section 3.3 can be extended and optimized in different ways: Section 3.5 describes some of these optimizations on the previous examples. Finally, a preliminary experimentation of the technique, that shows promising time savings, is presented in Section 3.6, while conclusions and future work are given in Section 3.7.

3.2 Permutation problems

A permutation of tuples in a r-ary relation R can be viewed as a new $2r$-ary relation P with components in R defining a linear order among its tuples (cf. Section 1.2). Equivalently, we can identify a permutation with a bijective function from the set of tuples in R to the set of bounded integers in the interval $[1, |R|]$, where $|R|$ is the number of tuples that belong to relation R. For the sake of simplicity, in the following we use the second characterization. We already introduced bounded integers in Section 1.2, and observed that this does not affect the expressive power of ESO.

Formula (1.1) characterizes ESO specifications for search problems in general. However, since this chapter focuses on permutation problems only, we assume that specifications for them are given in the following specialized form:

$$\exists P^{(r+1)}, \vec{S'} \;\; permutation(P, R^{(r)}) \;\wedge\; \varphi(\vec{S}, \vec{R}), \tag{3.1}$$

where R is a r-ary relation of (or a relational expression over relations in) the instance schema $\vec{R}$, and $P^{(r+1)}$ is a distinguished guessed predicate of $\vec{S} = \{P\} \cup \vec{S'}$, whose extensions are supposed to be permutations (i.e., total orderings) of tuples in R, as the expression for $permutation(\cdot, \cdot)$ states:

$$\begin{aligned}
permutation(P^{(r+1)}, R^{(r)}) \;\doteq\; & \forall \vec{x}, i \; P(\vec{x}, i) \;\rightarrow\; R(\vec{x}) \wedge (i \in [1, |R|]) \wedge & (3.2)\\
& \forall \vec{x} \; \exists i \; P(\vec{x}, i) \wedge & (3.3)\\
& \forall i \; \exists \vec{x} \; P(\vec{x}, i) \wedge & (3.4)\\
& \forall \vec{x}, i, i' \; P(\vec{x}, i) \wedge P(\vec{x}, i') \rightarrow i = i' \wedge & (3.5)\\
& \forall \vec{x}, \vec{x'}, i \; P(\vec{x}, i) \wedge P(\vec{x'}, i) \rightarrow x = x'. & (3.6)
\end{aligned}$$

The expression for $permutation(\cdot, \cdot)$ forces extensions of P to be bijections among tuples in R and bounded integers in the interval $[1, |R|]$. In particular, (3.2) forces P to be a relation in $R \times [1, |R|]$, while (3.3–3.6) force P to be, respectively, total, surgective, monodrome (i.e., a function), and injective (we call the last constraint *all-different*).

It is worth noting that every set of three of the constraints (3.3–3.6) entails the fourth, since the size of the domain of the function encoded by P is equal to the size of its codomain. To this end, from now on we ignore constraint (3.4).

Example 3.2.1 (The Hamiltonian path (HP) problem (62, Prob. GT39, p. 199)). *Given a graph as input, this* NP*-complete problem amounts to decide whether there exists a path in the graph that touches every node exactly once.*

This problem can be formulated in ESO different ways. As an example, in Example A.2.1 we give a possible specification for HP. On the other hand, here we give a formulation as a permutation problem: an HP is a permutation of the graph nodes such that every node is linked by an edge to the next one, with respect to the order given by the permutation.

In particular, assuming that the input graph is encoded in relations $node(\cdot)$ *and* $edge(\cdot, \cdot)$*, a specification for HP as permutation problem is as follows:*

$$\begin{aligned}&\exists P^{(2)}\ \mathit{permutation}(P, \mathit{node})\ \wedge\\ &\qquad \forall v, v', i\ P(v,i) \wedge P(v', i+1)\ \rightarrow\ \mathit{edge}(v,v').\end{aligned} \tag{3.7}$$

Constraint (3.7) *(i.e., the φ part of the specification, according to formula* (3.1)*) imposes the existence of an edge from any node to its successor. Figure 3.1(a) shows an instance and a solution of the HP problem.*

In what follows we present a reformulation technique that applies to the class of permutation problems characterized by the fact that the $\varphi(\vec{S}, \vec{R})$ part of the specification (3.1) contains constraints that bind an element of the permutation to the next or previous one only. It can be checked that the specification given in Example 3.2.1 belongs to this class.

To simplify the presentation of our reformulation technique, we will initially use the HP problem as a running example. However, it is worth noting that our technique can be used with any specification in the class defined above. To this end, in Section 3.4 we show other examples of problem specifications for which our technique works.

3.3 Reformulating Hamiltonian path

Let us consider the specification in Example 3.2.1, and let us ignore (3.6) (the all-different constraint) in the definition of *permutation*$(\cdot,\cdot)$. As a result, relation P can map several tuples of relation *node*$(\cdot)$ into the same integer, thus inducing only a partial order on the graph nodes. We say that nodes have been divided into *groups.* As an example, Figure 3.1(b) shows a situation in which the two nodes in group 5 can be visited in any order, one of them leading to the solution in Fig. 3.1(a). More formally, the problem can be *abstracted* from the level of nodes to the level of groups of nodes.

If, coherently with the relaxation of (3.6), constraint (3.7) is extended to handle pairs of (distinct) nodes in the same group, i.e., to:

$$\begin{aligned}&\forall v, v', i\ \ P(v,i) \wedge P(v', i+1)\ \rightarrow\ \mathit{edge}(v,v')\ \wedge\\ &\forall v, v', i\ \ P(v,i) \wedge P(v', i) \wedge v \neq v'\ \rightarrow\ \mathit{edge}(v,v'),\end{aligned} \tag{3.7'}$$

we obtain, except for some technicalities we will discuss next, a possible reformulation of the original HP specification, since, for every input instance, a solution of the reformulated specification can be translated in polynomial time (actually in linear time in the size of the graph) to a solution of the original problem (via a post-processing stage), by choosing an arbitrary order of nodes in the same group.

It can be observed that constraint (3.7') has a slightly different meaning with respect to (3.7):

1. It forces each group to be "strongly" connected to the next one (by forcing *every* node of every group to be linked to *every* node of the next one);
2. It forces subgraphs induced by groups of nodes to be *cliques*.

The PostProcessing stage of the computation (cf. Figure 2.2) is the same as enforcing the *delayed* constraint (3.6), i.e., forcing the partial order on nodes to be total.

Hence, since (3.6) can be ignored (in a first step) regardless of the input instance, we call it a *safe-delay* constraint. However, a *modification* of the remaining constraints (e.g., (3.7)) is needed in this case to guarantee the correctness of the reformulation.

The reformulated specification for the HP problem is thus as follows:

$$\exists P^{*(2)}\ \mathit{permutation}_{\text{w/o (all-diff)}}(P^*, \mathit{node}) \wedge \tag{3.8}$$

$$\begin{aligned} &\forall v, v', i\ \ P^*(v,i) \wedge P^*(v', i+1) \ \rightarrow\ \mathit{edge}(v,v') \wedge \\ &\forall v, v', i\ \ P^*(v,i) \wedge P^*(v', i) \wedge v \neq v' \ \rightarrow\ \mathit{edge}(v,v'), \end{aligned} \tag{3.9}$$

where $\mathit{permutation}_{\text{w/o (all-diff)}}(P^*, \mathit{node})$ is obtained by ignoring the all-different constraint (3.6) from the definition of $\mathit{permutation}(P^*, \mathit{node})$, cf. (3.2–3.6).

Constraints (3.3) and (3.5) in the expression for $\mathit{permutation}_{\text{w/o (all-diff)}}(P^*, \mathit{node})$ partition the graph nodes into groups, opportunely linked by edges (3.9) to allow the traversal of the whole graph. Since groups are cliques, paths linking all nodes in them surely exist, and, for every input instance, any extension of P^* that is a solution of the reformulated specification, can be transformed in an extension of P that is a valid solution of the original one, by choosing one such sub-path for every group in an arbitrary way.

However, some technicalities must be considered: in particular, the expression "$i+1$" in constraint (3.9) must be read as *the successor* group with respect to i, since the sequence of group numbers can have "holes" (i.e., P^* can map nodes to, e.g., groups 1 and 3, but not to 2). To deal with these issues in a simple way, we add a new constraint to the definition of $\mathit{permutation}_{\text{w/o (all-diff)}}(\cdot,\cdot)$, saying that the sequence of group numbers does not have holes (we call it the *compactness* constraint). It is worth noting that this constraint preserves satisfiability and can be removed by a different encoding of the search space (e.g., by avoiding integers and encoding it as a partial order on nodes by means of a *successor* relation).

Figure 3.1(b) shows a solution of the reformulated problem for the same instance in Figure 3.1(a), that leads to a set of solutions of the original one.

In Section 3.5 we will discuss two optimization of this reformulation technique, that lead to larger groups of nodes.

3.4 Further examples

In order to show how this technique can be applied to different problems in the class defined in Section 3.1, in the following we present ESO specifications for some of them: the *Permutation flow-shop* and the *Mono-dimensional tiling* problems.

Example 3.4.1 (The Permutation flow-shop (PFS) problem (62, Prob. SS55, p. 241)). *Given a set M of machines, a set J of jobs, each one consisting of a sequence of $|M|$ tasks (the i-th one to be executed on the i-th machine) of given positive durations, and an overall time deadline (makespan) D to be met, this* NP*-complete problem amounts to decide whether a permutation P of the jobs, and a starting time for each task of each job exists, such that:*

1. *For every job j and for every machine m, task of j at m is executed after task of job $j-1$ at m;*

2. *For every job, the execution of its task on any machine starts only after the task on the previous machine terminates;*

3. *Each machine executes at most one task at a time;*

4. *No preemption is allowed (i.e., a task must be terminated once started);*

5. *The execution of all jobs terminates before time D.*

Input to this problem is assumed to be given by relations $M(\cdot)$ listing the machines, $J(\cdot)$ listing jobs, $dur(\cdot,\cdot,\cdot)$ encoding duration of tasks (a tuple $\langle j, m, d\rangle$ in relation $dur(\cdot,\cdot,\cdot)$ means that the task of job j to be executed on machine m has duration d), and $Time(\cdot)$ encoding all time instants from 0 to D.

For the sake of simplicity, we assume that tuples in relations M and $Time$ are ordered (e.g., represented by integers), and the function "+" on elements in the domain of relation $Time$ is defined.

An ESO specification for the PFS problem is as follows:

$$\begin{aligned}
\exists P^{(2)}, S^{(3)}\ & permutation(P, J)\ \wedge \\
& totalFunction(S, \langle J, M\rangle, Time)\ \wedge && (3.10)\\
& \forall i\ \forall j, m, t\ \forall j', t', d'\ \big(P(j,i) \wedge P(j', i-1) \wedge S(j,m,t)\ \wedge && (3.11)\\
& \qquad S(j',m,t') \wedge\ dur(j',m,d')\big)\ \rightarrow\ (t \geq t' + d')\ \wedge \\
& \forall j, m, t, t_{prv}, d_{prv}\ \big(S(j,m,t) \wedge S(j, m-1, t_{prv}) \wedge && (3.12)\\
& \qquad dur(j, m-1, d_{prv})\big)\ \rightarrow\ (t \geq t_{prv} + d_{prv})\ \wedge \\
& \forall j, m, t, d\ S(j,m,t) \wedge dur(j,m,d) \rightarrow Time(t+d). && (3.13)
\end{aligned}$$

Guessed predicate $P^{(2)}$ encodes a total order among jobs, while $S^{(3)}$ represents the starting time for each task of each job: tuple $\langle j, m, t\rangle$ in S means that task m of job j is scheduled to start at time t.

From the specification above, S is forced to be a total function (cf. (3.10)*) from pairs in $J \times M$ to tuples in $Time$, i.e., it assigns a unique starting time (up to the horizon D) to every task of every job (we omit the definition of $totalFunction(\cdot,\cdot,\cdot)$, that can be derived from the other expressions in Appendix A); we just say that the second argument is the domain, while the third is the codomain of the mapping.*

As for the other constraints, (3.11) *forces the starting times assignment to let each machine execute at most one job at each time instant, while* (3.12) *forces each task of any job to start only when the previous task of the same job terminates. Finally,* (3.13) *forces the scheduling to meet the global deadline D.*

By applying our reformulation technique to the PFS problem, we allow different jobs to be assigned by P to the same integer, i.e., that are grouped together to form a unique job. For every input instance, once a solution of the reformulated specification is obtained, any arbitrary order among jobs grouped together leads to a valid solution of the original problem (cf. Figures 3.1(c) and (d)). Anyway, as we will briefly discuss in Section 3.5, a relaxation of the deadline can be used to optimize the reformulation.

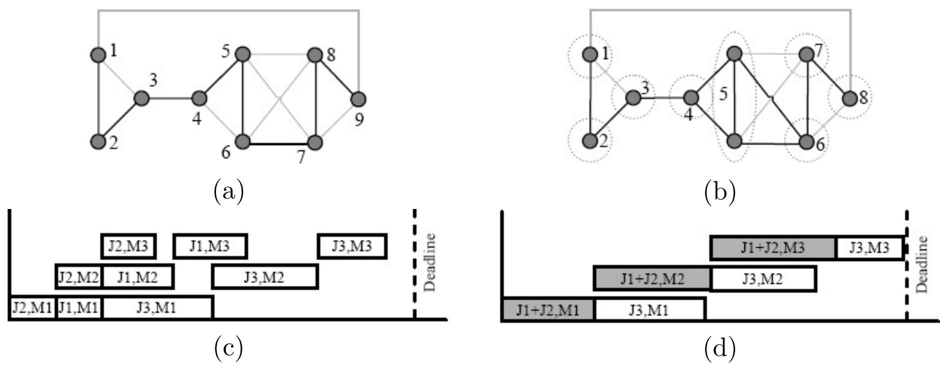

Figure 3.1: An instance and a solution for HP (a) and PFS (3 jobs, 3 machines) (c), and a solution of the reformulated specification for the same instance ((b) and (d)), subsuming the one in (a) and (c), respectively.

Example 3.4.2 (The Mono-dimensional tiling (1DT) problem (62, A variant of Prob. GP13, p. 257)). *Given a set C of colors and a set of tiles with their left and right sides colored with colors in C, this* NP*-complete problem amounts to decide whether a permutation P of the tiles exists, such that, for every pair of tiles one next to the other with respect to the order given by the permutation, colors on adjacent sides coincide.*

Input to this problem is assumed to be given by relations $C(\cdot)$ for colors and $Tile(\cdot)$ for tiles, and colors for the left and right side of each tile are represented by relation $TileColors(\cdot,\cdot,\cdot)$ (a tuple $\langle t,l,r\rangle$ in $TileColors$ means that tile t has the left side colored in l and the right one in r, where both l and r belong to C).

An ESO specification for the 1DT problem is as follows:

$$\begin{aligned}\exists P^{(2)}\ & permutation(P, Tile) \wedge \\ & \forall i\ \forall t,l,r,\ \forall t',l',r'\ P(t,i) \wedge P(t',i+1) \wedge \\ & \quad TileColors(t,l,r) \wedge TileColors(t',l',r') \ \rightarrow\ (r = l'). \end{aligned} \tag{3.14}$$

The straightforward application of our reformulation technique for this specification leads to different tiles that can be assigned to the same position number in the sequence, i.e., grouped together to form a block that behaves just like a single tile. Any ordering of tiles in the same block can be used to obtain a valid solution of the original problem, for every input instance, and for every solution of the reformulated specification. However, by the other constraints, this reformulation forces all tiles in a block to be equal to each other, and such that the color on their left side is the same as the one on their right side. Again, we will briefly describe an optimized reformulation for this specification in Section 3.5.

3.5 Optimizations and extensions

In this section we discuss some alternatives to optimize specifications that are obtained by the application of our technique.

Let us consider again the specification of the HP problem (cf. Example 3.2.1), and its reformulation (3.8–3.9), that we refer to as *Reformulation 1* (cf. also Figure 3.1(a) and (b)).

From Figure 3.1(b), we can observe how the price for allowing the abstraction from the level of nodes to the level of groups of nodes (i.e., (3.9)) is very high, since it imposes *every* node of a group to be linked to *every* node of the next one. As a consequence, groups are usually very small.

A first optimization (which we refer to as *Reformulation 2* of HP) is to force every node of a group to be linked to *at least one* (and not to every) node of the next group. This yet guarantees that every permutation of nodes in a group is a good sub-path of the whole path. This leads to the following specification:

$$\exists P^* \ \mathit{permutation}_{\text{w/o (all-diff)}}(P^*, \mathit{node}) \ \wedge \tag{3.15}$$

$$\begin{aligned} &\forall v, i \ \neg \mathit{last}(P^*, i) \ \rightarrow \ \exists v' \ \ P^*(v,i) \wedge P^*(v', i+1) \rightarrow \mathit{edge}(v, v') \ \ \wedge \\ &\forall v, v', i \ \ P^*(v,i) \wedge P^*(v',i) \wedge v \neq v' \ \ \rightarrow \ \ \mathit{edge}(v,v'), \end{aligned} \tag{3.16}$$

where

$$\mathit{last}(P^*, i) \ \doteq \ \exists v^i \ P^*(v^i, i) \ \wedge \neg \exists v^{i+1} \ P^*(v^{i+1}, i+1),$$

i.e., group i is the last one with respect to the order given by P^*.

Figure 3.2(a) shows a possible solution of the reformulated problem, subsuming the one in Figure 3.1(b).

Actually, we can make this intuition much more reasonable, observing that what we really want is to go from one group to the next without touching a node twice. So, if we start traversing group i from node v, we need an edge from a node $v' \neq v$ in group i to some node in group $i+1$ (if present). This allows us to choose a path for group i starting from v and ending in v'. Such a path surely exists, since groups are cliques (cf. Fig. 3.2(b)). This (simplified) constraint can be formalized in the following way: each group i needs to have at least two distinct nodes, v^i_{in} and v^i_{out}, such that:

- An edge exists from $v^{(i-1)}_{out}$ to v^i_{in}, for every $i > 1$;
- An edge exists from v^i_{out} to $v^{(i+1)}_{in}$, for every $i < k$, with k being the last group.

We call this new reformulation for the HP problem *Reformulation 3*, and its formal specification is as follows:

$$\exists P^{*(2)} \ \mathit{permutation}_{\text{w/o (all-diff)}}(P^*, \mathit{node}) \ \wedge \tag{3.17}$$

$$\begin{aligned} &\forall i \ \exists v^i_{in}, v^i_{out} \ \ P^*(v^i_{in}, i) \ \wedge \ P^*(v^i_{out}, i) \ \wedge v^i_{in} \neq v^i_{out} \ \wedge \\ &\quad (i > 1) \rightarrow \exists v^{i-1}_{out} \ P^*(v^{i-1}_{out}, i-1) \wedge \mathit{edge}(v^{i-1}_{out}, v^i_{in}) \ \wedge \\ &\quad \neg \mathit{last}(P^*, i) \rightarrow \exists v^{i+1}_{in} \ P^*(v^{i+1}_{in}, i+1) \wedge \mathit{edge}(v^i_{out}, v^{i+1}_{in}). \end{aligned} \tag{3.18}$$

In general, for any instance, the following inclusion relationship holds:

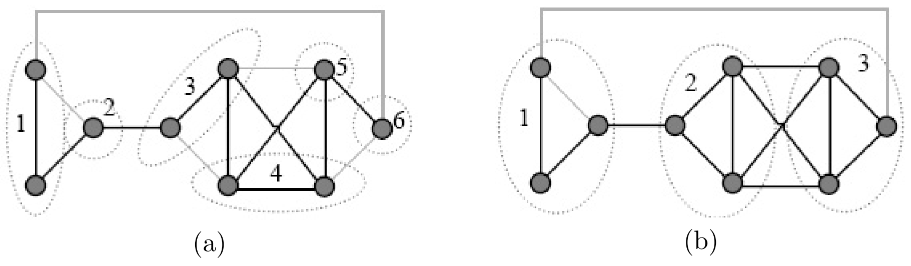

Figure 3.2: A solution of Reformulation 2 (a) and 3 (b) of HP for the instance of Fig. 3.1(a), and subsuming solution of Fig. 3.1(b).

$$\{\textit{Sols of orig. spec}\} \subseteq \{\textit{Sols of Ref. 1}\} \subseteq \{\textit{Sols of Ref. 2}\} \subseteq \{\textit{Sols of Ref. 3}\}.$$

As for PFS, it can be observed that allowing several jobs to be grouped together leads to longer schedulings (cf. Figure 3.1(d)). To contrast this phenomenon, a possible optimization of our technique is to augment the makespan, yet guaranteeing that, for every solution of the reformulated problem, every ordering of jobs in the same group leads to a valid solution of the original problem (with the original deadline). We are currently investigating different alternatives and heuristics for the makespan augmentation, from the simplest one, that relies on the minimum and maximum duration of tasks of each job, to more complex expressions, that rely also on the particular grouping of jobs guessed.

Also the reformulated specification for 1DT can be optimized, in a similar way of that for HP, i.e., by allowing that at least one ordering of tiles in the same group (and not every one) leads to a good sub-tiling that correctly interfaces with the rest of the solution. Furthermore, also a limitation on the group size is possible, to let this check to be first-order, and so polynomial time.

3.6 Experimentation

In order to check the effectiveness of this technique, we made a preliminary experimentation on several instances for the specifications described above. In particular, experiments involved both the DPLL-base SAT solver EQSATZ (86) and the state-of-the-art CP engine Ilog SOLVER, invoked through OPLSTUDIO.

In particular, we made the following experiments:

- *Hamiltonian path*: random instances (graphs between 16 and 24 nodes) solved with both EQSATZ (and an ad-hoc program for the instantiation), and Ilog SOLVER;
- *Permutation flow shop*: benchmark instances taken from the OR-Library solved with Ilog SOLVER;
- *Mono-dimensional tiling*: random instances (from 9 to 11 tiles and different values for the number of colors), created with an home-made generator and solved with Ilog SOLVER.

Experiments were executed on an Intel 2.4 GHz Xeon bi-processor computer and on a PentiumII.

As for HP, we solved all three reformulated specifications. For what concerns the SAT-based experimentation, consistent time savings (from seconds to several minutes, and up to 95%) have been obtained for almost all instances. On the other hand, Ilog SOLVER seems to be negatively affected by reformulation on this problem: in particular, solving Reformulation 1 is rarely faster than solving the original specification, and, moreover, the solver was often unable to solve Reformulations 2 and 3 in reasonable time. A similar behavior has been observed for 1DT. As for future work, we plan to use a SAT solver for this problem, to test the effectiveness of the technique in this case.

On the contrary, the application of the proposed reformulation to PFS lead to consistent time savings to the SOLVER specification, especially on negative instances. In particular, for many of them, deciding that the reformulated specification had no solutions needed very few seconds, while the solver did not terminate in 30 minutes with the original specification.

3.7 Discussion and future research directions

In this chapter, we extended reformulation by safe-delay, presented in Chapter 2, in order to deal with a class of permutation problems. We identified the *all-different* constraint as safe-delay, and proposed a reformulation technique that allows to ignore its evaluation in a first-stage, with the goal of improving the performances of the solving engine. Once a solution of the new specification has been found, it is possible to find a valid solution of the original problem in an efficient (i.e., polynomial time) way, without further search. Having a larger set of solutions, the reformulated specification is thus likely to be more efficiently solvable for some classes of solvers.

The main difference between this technique and that of Chapter 2 is that a modification of the remaining constraints of the specification can be necessary to guarantee the correctness of the reformulation. We characterized a class of permutation problems for which the reformulation works, and presented encouraging experimental results that show that this technique improves solving times on several classes of instances for different problems and different solvers.

Finally, it is worth noting that, even if the above reformulations seem much more complex than those in Chapter 2, they are absolutely *syntax-based*, and therefore are completely automatable.

As for future work, we plan to characterize the presented reformulation technique in a formal way, providing formal proofs of the correctness of the transformation. Secondly, we plan to further analyze the various extensions and optimizations of our technique discussed in Section 3.5, e.g., by providing a better relaxation of the deadline in the reformulation of PFS. A more exhaustive experimentation, in the style of that of Section 5.5 is also planned. As ultimate goal, it would be very interesting to understand how to make the system able to automatically perform these kind of optimizations, by modifying the other constraints.

Preliminary results of this chapter have been published in (90).

Chapter 4

Detecting and breaking symmetries on problem specifications

4.1 Introduction

In this chapter we tackle the reformulation problem in presence of symmetries in the specification, that have been widely recognized to be one of the major obstacles for the efficient solving of combinatorial and constraint satisfaction problems.

Much work has been already done, and a wide literature is nowadays available on how symmetries can be detected and exploited, with the aim of greatly reducing the size of the search space. Four are the main approaches that have been followed by the research community to deal with symmetries:

1. Imposing additional constraints to the model of the problem, which are satisfied only for one of the symmetrical points in the search space, cf., e.g., (107; 39; 110; 116; 119; 54);

2. Introducing additional constraints during the search process, to avoid the traversal of symmetrical points, cf., e.g., (11; 4; 63; 56; 52; 108);

3. Defining a search strategy able to break symmetries as soon as possible (e.g., by first selecting variables involved in the greatest number of local symmetries), cf., e.g., (95);

4. Isolating subclasses of CSPs for which particular search strategies can be used in order to efficiently break their symmetries (cf., e.g., tractability of symmetry breaking for CSPs with various form of interchangeability (125)).

We follow the first approach, but, differently from other works in the literature (e.g., (39; 110; 54)), we attack the problem at the logical level of the specification. In fact, in many cases, symmetries arise from the problem structure, and not from the particular instance considered. Hence, they can be very easily detected by humans (cf. Section 4.5) at the specification level, and convenient symmetry-breaking formulae can be added to the specification itself. Furthermore, since specifications are logical formulae, computer tools can be used to automatically or semiautomatically detect and break symmetries in complex cases (cf. Chapter 6).

Finally, detecting and breaking symmetries at the specification level does not rule out the possibility to use symmetry-breaking techniques at the instance level, in order to deal also with additional symmetries that arise from the problem instance. As an example, since some systems generate a SAT instance, e.g., (26), or an instance of integer linear programming, e.g., (124), it is possible to do symmetry breaking on such instances, using existing techniques, cf. e.g., (40).

The outline of this chapter is as follows: in Section 4.2, after recalling basic definitions of symmetries on CSPs, we lift to the specification level, defining the concepts of symmetry of a specification and of symmetry-breaking formula. Then, in Sections 4.3 and 4.4, we show, respectively, how to detect and break such symmetries, clarifying the new definitions by showing, in Section 4.5, some specifications: Graph 3-coloring, Not-all-equal Sat, Social golfer, and BIBD generation problems. For all of them we detect and break symmetries. Finally, Section 4.6 presents experimental results of our symmetry-breaking techniques on some of those examples, using state-of-the-art linear and constraint programming solvers CPLEX and SOLVER, invoked through OPLSTUDIO, while Section 4.7 draws conclusions and discusses future work.

4.2 Basic definitions

In this section we give the definitions of *transformation* and *symmetry* of a problem specification. These definitions can be obtained as a natural lifting of the respective definitions in the context of Constraint Satisfaction Problems (CSPs). To this end, we briefly recall these standard definitions, before focusing on the specification level.

Definition 4.2.1 (Constraint satisfaction problem (CSP)). *A CSP is a tuple $\langle\vec{V},\vec{D},\vec{C}\rangle$, where $\vec{V}=\{X_1,\ldots,X_n\}$ is a set of variables, $\vec{D}=\{D_1,\ldots,D_n\}$ is a set of finite domains, one for each variable, $\vec{C}=\{C_1,\ldots,C_m\}$ is a set of constraints on $\vec{V}$ and $\vec{D}$, i.e., a set of relations $C_i(X_{i_1},\ldots,X_{i_k}\}\subseteq D_{i_1}\times\cdots\times D_{i_k}$, for every $i\in[1..m]$, containing those configurations of assignments allowed by the constraint.*

Given a CSP π, we can transform it in a new CSP π' by exchanging variables and/or domains order. A CSP transformation is defined in the following way:

Definition 4.2.2 (CSP transformation). *Given a CSP $\pi=\langle\vec{V},\vec{D},\vec{C}\rangle$, with $\vec{V}=\{X_1,\ldots,X_n\}$ and $\vec{D}=\{D_1,\ldots,D_n\}$, a transformation of π is a set of bijections $\sigma_0,\sigma_1,\ldots,\sigma_n$ such that:*

$$\begin{aligned}\sigma_0 &: \vec{V}\to\vec{V}\\ \textit{for every } i\in[1,n],\ \sigma_i &: D_i\to D_{index(\sigma_0(X_i))},\end{aligned}$$

where $index()$ is a function that returns the index of the argument variable.

It is worth noting that a transformation is defined on $\vec{V}$ and $\vec{D}$ only, and not on $\vec{C}$.

Some interesting specializations of Definition 4.2.2 do exists. They apply also to forthcoming Definition 4.2.3. In particular, we have:

Variable transformation:

- $\sigma_1, \ldots, \sigma_n$ are the identity function;

Value transformation:

- σ_0 is the identity function;

Uniform value transformation:

- σ_0 is the identity function;
- $\sigma_1 = \sigma_2 = \cdots = \sigma_n$;
- $D_1 = D_2 = \cdots = D_n$.

Intuitively, a variable transformation exchanges only the order of variables, leaving the domains unchanged. On the other hand, a value transformation does not modify the variable order. In what follows, we focus on uniform value transformations, in which domain values for each variable are swapped *uniformly.*

As widely described in the literature, symmetries of a CSP are transformations that map solutions into solutions:

Definition 4.2.3 (CSP symmetry (94)). *Given a CSP $\pi = \langle \vec{V}, \vec{D}, \vec{C} \rangle$, a symmetry on π is a CSP transformation on $\langle \vec{V}, \vec{D} \rangle$ which preserves constraints: an assignment $\tau = \{X_1 = v_1, \ldots, X_n = v_n\}$ to variables in $\vec{V}$ satisfies every constraint in $\vec{C}$ if and only if $\sigma(\tau) = \{\sigma_0(X_1) = \sigma_{index(\sigma_0(X_1))}(v_1), \ldots, \sigma_0(X_n) = \sigma_{index(\sigma_0(X_1))}(v_n)\}$ does.*

We now focus on transformations and symmetries on specifications. In what follows, we initially focus on problem specifications which, like the one of Example 1.2.2, have only *monadic* guessed predicates. The reason for this limitation is that, in this way, we have a neat conceptual correspondence between the guessed predicates and *domain values* of a CSP.

Specifications with non-monadic guessed predicates can be handled by unfolding non-unary predicates and exploiting finiteness of the input database, as already argued in Section 1.2 and Subsection 2.2.3. Forthcoming Example 4.5.3 shows how our definitions can be used for non-monadic predicates.

In the context of ESO problem specifications, we give the following definition:

Definition 4.2.4 (Uniform value transformation (UVT) of a specification). *Given a problem specification $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$, with $\vec{S} = \{S_1, \ldots S_n\}$, S_i monadic for every i, and input schema $\vec{R}$, a* uniform value transformation (UVT) *for ψ is a mapping $\sigma : \vec{S} \rightarrow \vec{S}$, which is total and onto, i.e., defines a permutation of guessed predicates in $\vec{S}$.*

The term "uniform value transformation" in Definition 4.2.4 is used because swapping monadic guessed predicates is conceptually the same as uniformly exchanging domain values in a CSP.

From here on, given ϕ and σ as in the above definition, ϕ^σ is defined as

$$\phi[S_1/\sigma(S_1), \ldots, S_n/\sigma(S_n)],$$

i.e., ϕ^σ is obtained from ϕ by uniformly substituting every occurrence of each guessed predicate with the one given by the transformation σ. Analogously, ψ^σ is defined as $\exists \vec{S}\ \phi^\sigma(\vec{S}, \vec{R})$.

We now define when a UVT is a symmetry for a given specification.

Definition 4.2.5 (Uniform value symmetry (UVS) of a specification). *Let $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$ be a specification, with $\vec{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1, n]$, and input schema $\vec{R}$, and let σ be a UVT for ψ. Transformation σ is a* uniform value symmetry (UVS) *for ψ if every extension for $\vec{S}$ which satisfies ϕ, satisfies also ϕ^σ and vice versa, regardless of the input instance, i.e., for every extension of the input schema $\vec{R}$.*

Note that every CSP obtained by instantiating a specification with UVS σ has at least the corresponding uniform value symmetry. The above definition could in principle be stated with a different pattern of quantifiers, as an example, we could require equivalence only for *some* instances, instead that for each instance. Generalization of the definition is left for future research.

4.3 Detecting symmetries on specifications

In this section we face with the problem of detecting UVSs by reasoning on specifications. We show that checking whether a UVT is a UVS reduces to checking equivalence of two first-order formulae. As a consequence, this task can be performed by a first-order theorem prover (cf. forthcoming Chapter 6). Proofs of the following theorems are delayed to Appendix C.

Theorem 4.3.1. *Let $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$ be a problem specification, with $\vec{S} = \{S_1, \ldots S_n\}$, S_i monadic for every i, and σ a UVT for ψ. Transformation σ is a UVS for ψ if and only if $\phi \equiv \phi^\sigma$.*

The following theorem shows that also the converse reduction can be proven, thus implying that checking whether a UVT is a UVS is not decidable.

Theorem 4.3.2 (Undecidability of the UVS checking problem). *Given a specification ψ and a UVT σ for it, the problem of checking whether σ is a UVS for ψ is undecidable.*

Example 4.3.1 (Graph 3-coloring: Example 1.2.2, continued). *Three UVTs are as follows:*

- $\sigma^{R,G}$ *s.t.* $\sigma^{R,G}(R) = G$, $\sigma^{R,G}(G) = R$, $\sigma^{R,G}(B) = B$;
- $\sigma^{R,B}$ *s.t.* $\sigma^{R,B}(R) = B$, $\sigma^{R,B}(G) = G$, $\sigma^{R,B}(B) = R$;
- $\sigma^{G,B}$ *s.t.* $\sigma^{G,B}(R) = R$, $\sigma^{G,B}(G) = B$, $\sigma^{G,B}(B) = G$.

It is easy to observe that, for each of the above transformations, formulae $\phi^{\sigma^{R,G}}$, $\phi^{\sigma^{R,B}}$*, and* $\phi^{\sigma^{G,B}}$ *are all equivalent to* ϕ*, because clauses of the former ones are syntactically equivalent to clauses of* ϕ *and vice versa. This implies, by Theorem 4.3.1, that they are all UVSs.*

To play the devil's advocate, we consider a modification of the problem of Example 1.2.2, and show that only one of the UVTs in Example 4.3.1 is indeed a UVS in the new problem.

Example 4.3.2 (Graph 3-coloring with red self-loops (Example 2.2.1 continued)). *Let us consider UVTs described in Example 4.3.1:*

- $\sigma^{G,B}$*: it is yet a UVS, because the argument of Example 4.3.1 applies.*
- $\sigma^{R,G}$*: in this case,* $\phi^{\sigma^{R,G}}$ *is not equivalent to* ϕ *any more: as an example, for the input instance* $edge = \{(v,v)\}$*, the color assignment* $\overline{R},\overline{G},\overline{B}$ *such that* $\overline{R} = \{v\}, \overline{G} = \overline{B} = \emptyset$ *is a model for the original problem, i.e.,* $\overline{R},\overline{G},\overline{B} \models \phi(R,G,B,edge)$*. It is however easy to observe that* $\overline{R},\overline{G},\overline{B} \not\models \phi(R,G,B,edge)^{\sigma^{R,G}}$*, because* $\phi^{\sigma^{R,G}}$ *is verified only by color assignments for which* $\overline{G}(v)$ *holds. This implies, by Theorem 4.3.1, that* σ *is not a UVS.*
- $\sigma^{R,B}$*: it is not a UVS, because the same argument of the previous point applies.*

Theorem 4.3.1 suggests that the problem of detecting UVSs of an ESO specification ψ can in principle be performed in the following way:

1. Selecting a UVT σ, i.e. a permutation of guessed predicates in ψ;
2. Checking whether σ is a UVS, deciding whether $\phi \equiv \phi^{\sigma}$.

If ψ has n guessed predicates, there are $n!$ possibilities for point 1. As for point 2, we proved that UVS checking is not decidable (cf. Theorem 4.3.2). However, as we show in Chapter 6, first-order theorem provers and finite model finders can be used to perform automatically this task. The proposed algorithm, taken "as-is", may require $n!$ checks for point 2. However, in practice, n, i.e., the number of guessed predicates in the specification, is expected to be very small and, furthermore, good heuristics can be found for point 1. As an example, there may exist efficient techniques for detecting (in some cases, even with a deduction-free analysis) particular classes of symmetries (cf., e.g., (126)).

4.4 Breaking symmetries

As already mentioned in Section 4.1, once symmetries for a problem have been detected, several approaches can be used in order to exploit them. In what follows, we consider adding additional constraints to the problem specification, in order to *break* its symmetries, i.e., to wipe out from the solution space (some of) the symmetrical points. This kind of constraints are called *symmetry-breaking formulae*, and are defined as follows.

Definition 4.4.1 (Symmetry-breaking formula). *Let $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$, be a specification, with $\vec{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1, n]$, and input schema $\vec{R}$, and let σ be a UVS for ψ. A* symmetry-breaking formula *for ψ with respect to symmetry σ is a closed (except for $\vec{S}$) formula $\beta(\vec{S})$ such that the following two conditions hold:*

1. *Transformation σ is no longer a symmetry for $\exists \vec{S}\ \phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S})$:*

$$(\phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S})) \not\equiv (\phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S}))^{\sigma}; \tag{4.1}$$

2. *Every model of $\phi(\vec{S}, \vec{R})$ can be obtained by those of $\phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S})$ by applying symmetry σ:*

$$\phi(\vec{S}, \vec{R}) \models \bigvee_{\vec{\sigma} \in \sigma^*} (\phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S}))^{\vec{\sigma}}. \tag{4.2}$$

   where $\vec{\sigma}$ is a sequence (of finite length ≥ 0) over σ, and, given a first-order formula $\gamma(\vec{S})$, $\gamma(\vec{S})^{\vec{\sigma}}$ denotes $(\cdots(\gamma(\vec{S})^{\sigma})\cdots)^{\sigma}$, i.e., σ is applied $|\vec{\sigma}|$ times (if $\vec{\sigma} = \langle\rangle$, i.e., the empty string, then $\gamma(\vec{S})^{\vec{\sigma}}$ is $\gamma(\vec{S})$ itself).

If $\beta(\vec{S})$ matches the above definition, then we are entitled to solve the problem $\exists \vec{S}\ \phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S})$ instead of the original one $\exists \vec{S}\ \phi(\vec{S}, \vec{R})$. In fact, point 1 in the above definition states that formula $\beta(\vec{S})$ actually breaks σ, since, by Theorem 4.3.1, σ is not a symmetry of the rewritten problem. Furthermore, point 2 states that every solution of $\phi(\vec{S}, \vec{R})$ can be obtained by repeatedly applying σ to some solutions of $\phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S})$. Hence, all solutions are preserved in the rewritten problem, up to symmetric ones.

It is worthwhile noting that, even if in formula (4.2) $\vec{\sigma}$ ranges over the (infinite) set of finite-length sequences of 0 or more applications of σ, this actually reduces to sequences of length at most $n!$, since this is the maximum number of successive applications of σ that can lead to all different permutations. Moreover, we observe that the inverse logical implication always holds, because σ is a UVS, and so $\phi(\vec{S}, \vec{R})^{\sigma} \equiv \phi(\vec{S}, \vec{R})$. Finally, from point 1 it follows that a problem specification for which there are no satisfiable instances, i.e., which first-order part $\phi \equiv \bot$, does not have symmetries. We do not further consider such case, since it would evidence a bug in the model.

We observe that breaking a symmetry is sound, i.e., it preserves at least one solution, as shown by the following theorem:

Theorem 4.4.1 (Symmetry-breaking formulae preserve satisfiability). *Let $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$ be a problem specification with $\vec{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1, n]$, and input schema $\vec{R}$, and let σ be a UVS for ψ. Furthermore, let $\beta(\vec{S})$ be a symmetry-breaking formula for ψ with respect to σ. For each input instance $\mathcal{I}$, i.e., for each extension of the input schema $\vec{R}$, if $\exists \vec{S}\ \phi(\vec{S}, \mathcal{I})$ has solutions, then also $\exists \vec{S}\ \phi(\vec{S}, \mathcal{I}) \wedge \beta(\vec{S})$ has solutions.*

The proof is delayed to Appendix C. Given a problem specification ψ and a UVS σ, in general there exist several formulae $\beta(\vec{S})$ that are symmetry-breaking for ψ with respect to σ. Hence, a measure of the effectiveness of a symmetry-breaking formula can be defined. In what follows, we consider the effectiveness of a symmetry-breaking formula higher when $(\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S}))$ and $(\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S}))^\sigma$ have few common models, i.e., the less models in the conjunction $(\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S})) \;\wedge\; (\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S}))^\sigma$, the better. However, it must be observed that this measure of the effectiveness of a symmetry-breaking formula focuses only on how completely the formula breaks the symmetry, and not on efficiency issues like its amenability to propagation. In the next section we see several examples of UVSs and different breaking formulae for them.

Definition 4.4.1 deals with breaking a single symmetry, and it can be generalized in order to break *multiple* symmetries simultaneously. The definition, omitted for simplicity, deals with a set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ of UVSs. The first condition must be repeated for all symmetries in Σ, and the second one becomes:

$$\phi(\vec{S},\vec{R}) \models \bigvee_{\vec{\sigma} \in \Sigma^*} (\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S}))^{\vec{\sigma}} \tag{4.3}$$

where $\vec{\sigma} = \langle \sigma_{i_1}, \ldots, \sigma_{i_m} \rangle$ ($i_j \in [1,n]$ for each $1 \leq j \leq m$ and $m \geq 0$) is a finite sequence with elements in Σ. Thus, the above formula states that each solution of $\phi(\vec{S},\vec{R})$ can be obtained by applying some finite sequence of symmetries in Σ to some solutions of $\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S})$.

4.5 Examples of symmetry-breaking formulae

In this section we discuss some examples of well-known combinatorial problems that have many symmetries, show some of them using Theorem 4.3.1, and present symmetry-breaking formulae for them. In Appendix C we then show that formulae below respect both conditions of Definition 4.4.1.

Example 4.5.1 (Graph 3-coloring: Example 4.3.1, continued). *In the following, we present different symmetry-breaking formulae for this problem, referring to $\sigma^{R,G}$, $\sigma^{R,B}$, and $\sigma^{G,B}$.*

1. *Let us consider $\sigma^{G,B}$: a simple symmetry-breaking formula is:*

$$\beta_{sel}^{G,B}(R,G,B) \doteq G(\overline{v}) \vee R(\overline{v}), \tag{4.4}$$

that forces a selected node, say $\overline{v}$, not to be colored in blue.

It is worth noting that the simpler formula $G(\overline{v})$ is not a symmetry-breaking formula for $\sigma^{G,B}$, since it does not match condition (4.2) of Definition 4.4.1. To give the intuition for this fact, it suffices to observe that the symmetric assignment would color $\overline{v}$ in blue ($G(\overline{v})^{\sigma^{G,B}}$ is $B(\overline{v})$), so loosing all solutions that color $\overline{v}$ in red. Actually, the stronger formula $G(\overline{v})$ breaks two different symmetries, $\sigma^{G,B}$ and $\sigma^{G,R}$, and can be obtained as the logical "and" of (4.4) and $G(\overline{v}) \vee B(\overline{v})$.

2. *A different symmetry-breaking formula for the same symmetry is the following one:*

$$\beta^{G,B}_{least}(R,G,B) \ \doteq\ \forall Y\ B(Y) \ \rightarrow\ \exists X\ G(X) \wedge X < Y, \tag{4.5}$$

   where '$<$' is a (possibly pre-interpreted) total ordering on the graph nodes. This formula forces the least blue node to be greater than the least green one (thus, forcing the so-called lowest index ordering *on green and blue nodes).*

3. *A more complex symmetry-breaking formula for $\sigma^{G,B}$ is:*

$$\beta^{G,B}_{\leq}(R,G,B) \ \doteq\ |G| \leq |B|, \tag{4.6}$$

   that forces the number of green nodes to be less than or equal to the number of blue ones (this constraint can be written in ESO using standard techniques, cf. Section 1.2). In Appendix C we show that formula (4.6) *respects both conditions of Definition 4.4.1.*

   We note that $|G| < |B|$ is not a symmetry-breaking formula, since point 2 of Definition 4.4.1 is not satisfied: in fact adding such a constraint is obviously unsound, since some satisfiable instances may become unsatisfiable.

4. *Let us now consider the two symmetries $\sigma^{R,G}$ and $\sigma^{R,B}$: a multiple symmetry-breaking formula for them is:*

$$\beta^{R}_{\leq}(R,G,B) \ \doteq\ |R| \leq |G| \wedge |R| \leq |B|, \tag{4.7}$$

   that uses a partial order among the sets of nodes with the same color, R being the minimal element.

5. *The next formula breaks all three symmetries:*

$$\beta_{\leq}(R,G,B) \ \doteq\ |R| \leq |G| \wedge |R| \leq |B| \wedge |G| \leq |B|, \tag{4.8}$$

   by using a total order among colors, i.e., $|R| \leq |G| \leq |B|$. It is interesting to note that $\beta(R,G,B)$ can be obtained by composing formulae (4.6–4.7). Such a composition can be extended to the k-coloring problem for handling any number of colors.

Example 4.5.2 (Not-all-equal Sat, Example 1.2.3 continued). *UVT σ, such that $\sigma(T) = F$ and $\sigma(F) = T$ is a UVS for this problem, since ϕ^{σ} is clearly equivalent to ϕ. Symmetry-breaking formulae for σ are:*

$$\beta_{least}(T,F) \ \doteq\ \forall Y\ F(Y) \ \rightarrow\ \exists X\ T(X) \wedge X < Y, \tag{4.9}$$

where '$<$' is a (possibly pre-interpreted) total ordering among the variables, and:

$$\beta_{\leq}(T,F) \ \doteq\ |T| \leq |F|. \tag{4.10}$$

Example 4.5.3 (Social golfer (64, Prob. 10)). *Given a set of players, a set of groups, and a set of weeks, encoded in monadic relations player(·), group(·), and week(·) respectively, this problem amounts to decide whether there is a way to arrange a scheduling for all weeks in relation week, such that:*

- *For every week, players are divided into equal sized groups;*
- *Two different players don't play in the same group more than once.*

A specification for this problem (assuming the ratio $|player|/|group|$*, i.e., the group size, integral) is the following (*$Play(P, W, G)$ *states that player* P *plays in group* G *on week* W*):*

$$
\begin{aligned}
\exists Play\ \forall P, W, G\ Play(P, W, G)\ &\rightarrow\ player(P) \wedge week(W) \wedge group(G)\ \wedge && (4.11)\\
\forall P, W\ player(P) \wedge week(W)\ &\rightarrow\ \exists G\ Play(P, W, G)\ \wedge && (4.12)\\
\forall P, W, G, G'\ Play(P, W, G) \wedge Play(P, W, G')\ &\rightarrow\ G = G'\ \wedge && (4.13)\\
\forall P, P', W, W', G, G' & && (4.14)\\
(P \neq P' \wedge W \neq W' \wedge PLAY(P, W, G) &\wedge PLAY(P', W, G)) \rightarrow\\
\neg\,[Play(P, W', G') &\wedge Play(P', W', G')]\ \wedge\\
\forall G, G', W, W'\ group(G) \wedge group(G') \wedge week(W) &\wedge week(W')\ \rightarrow && (4.15)\\
|\{P : Play(P, W, G)\}| &= |\{P : Play(P, W', G')\}|.
\end{aligned}
$$

Constraints (4.11–4.13) force Play to be a total function assigning a group to each player on each week; moreover, (4.14) *is the* meet only once *constraint, while* (4.15) *forces groups to be of the same size (Appendix A shows how this constraint can be written in ESO).*

In order to highlight UVSs according to Definition 4.2.5, we need to substitute the ternary guessed predicate Play by means of, e.g., $|week| \times |group|$ *monadic predicates* $Play_{W,G}$ *(each one listing players playing in group* G *on week* W*). The above specification must be unfolded accordingly. We observe that, in this case, the number of monadic guessed predicates obtained as output of the unfolding actually depends on the input instance. However, we stress that in this case unfolding non-monadic guessed predicates is only a formal step in order to apply the definitions in this chapter, and has not to be performed in practice.*

On the unfolded specification, UVTs $\sigma_W^{G,G'}$*, swapping* $Play_{W,G}$ *and* $Play_{W,G'}$*, i.e., given a week* W*, and two groups* G *and* G'*, assign to group* G' *on week* W *all players assigned to group* G *on week* W*, and vice versa, are symmetries for the Social golfer problem. Intuitively, UVTs* $\sigma_W^{G,G'}$ *are UVSs for the unfolded specification because group renamings have no effect.*

After fixing arbitrary linear orders '<' on players and groups, each symmetry $\sigma_W^{G,G'}$ *(with* $G < G'$*) can be broken by, e.g., the following formula:*

$$
\begin{aligned}
&\beta_{least,W}^{G,G'}(Play_{W,G}, Play_{W,G'})\ \doteq\\
&\qquad \forall P'\ Play_{W,G'}(P')\ \rightarrow\ \exists P\ Play_{W,G}(P)\ \wedge\ P < P',
\end{aligned}
\qquad (4.16)
$$

that forces the least player in $Play_{W,G}$ *to have a lower order number (with respect to '<') than the least player in* $Play_{W,G'}$.

Similarly to Example 4.5.1, we can simultaneously break several of the above symmetries, by adding a constraint $\beta^{G,G'}_{least,W}(Play_{W,G}, Play_{W,G'})$ *for each week* W *and each pair* G, G' *of groups such that* G' *is the successor of* G *in the linear order '<', hence forcing, for each week, the least player in any group to have order number lower than the least player in the consecutive one. An implication of the above formulae is that the first player always plays in the first group.*

As another example, the following family of transformations –which we call $\sigma^{W,W'}$ *– are also symmetries for the original Social golfer specification: given two weeks* W *and* W'*, swap uniformly all groups in week* W *with the ones in week* W'*, and vice versa.*

Each symmetry $\sigma^{W,W'}$ *can be broken: in the following, we give only an intuitive description of a possible symmetry-breaking formula* $\beta^{W,W'}$*, once total orders '<' on weeks and on weekly assignments, i.e., on sets* $\{Play_{W,G_1}, \ldots, Play_{W,G_{|group|}}\}$ *have been fixed: given two assignments of players to groups* $\{Play_{W,G_1}, \ldots, Play_{W,G_{|group|}}\}$ *and* $\{Play_{W',G_1}, \ldots, Play_{W',G_{|group|}}\}$*, on weeks* W *and* W'*, they are swapped if and only if the first is greater than the second one.*

By breaking several of the aforementioned symmetries, we can add the following constraints, one for each pair W, W' *of weeks such that* W' *is the successor of* W *in the given linear order '<':*

$$\{Play_{W,G_1}, \ldots, Play_{W,G_{|group|}}\} < \{Play_{W',G_1}, \ldots, Play_{W',G_{|group|}}\} \tag{4.17}$$

saying that weekly assignments are in increasing order.

It is worth noting that, by combining the symmetry-breaking formulae for the two families of symmetries described above, we are able to get some of the symmetry-breaking constraints described in (120), like the one that arbitrarily fixes the assignment in the first week. Moreover, we can further fix one player (the one having the least order number) to be assigned always to the same group (the one having the least order number).

Finally, we observe that, by a different unfolding of the specification, it is possible to reproduce, by means of $|player| \times |week|$ *monadic predicates* $PLAY_{P,W}$*, the usual players/weeks matrix model of the problem, and encode the symmetry-breaking formula* β_{lex^2}*, that forces any assignment to respect the lexicographic ordering on both rows and columns of this matrix (54). In Appendix C we show the formal definition of* β_{lex^2} *and prove that it respects conditions of Definition 4.4.1.*

Example 4.5.4 (Balanced Incomplete Block Design (BIBD) generation (95)). *Given a set of* v *objects in a monadic relation object(·), a set of* b *blocks in a monadic relation block(·), with no tuples in common, plus positive integers* r, k, *and* λ*, this problem amounts to decide whether there is a way to arrange the* v *objects into the* b *blocks in such a way that each block contains exactly* k *distinct objects, each object occurs in exactly* r *different blocks, and every two distinct objects occur together in exactly* λ *blocks.*

A specification for this problem is quite similar to the one of Example 4.5.3, and also its symmetries are similar. We don't give here more details about this problem. Just as an example, it is always possible to swap the order of two blocks. Symmetry-breaking formulae can be built using the same techniques of Example 4.5.3.

From the above examples, it can be observed that the various symmetry-breaking formulae are often very similar to each other, usually following the same schema. Hence, they can be generalized in order to build a *library of templates*, from which generate candidate formulae to break UVSs for the problems at hand. An automated theorem prover can then be used to check whether some of the formulae in the library are actually symmetry-breaking (cf. Chapter 6).

As an example of template, a candidate formula for breaking symmetries that involve two guessed predicates is to force the size of the extension of the first predicate to be at most the size of the extension of the second one (cf., e.g., formulae (4.6) and (4.10)). This pattern for a symmetry-breaking formula can be generalized as follows: given a problem specification ψ of the kind $\exists\vec{S}\ \phi(\vec{S},\vec{R})$, with $\vec{S} = \{S_1, \dots S_n\}$, S_i monadic for every $i \in [1, n]$, and a UVS σ such that $\sigma(S_i) = S_j$, $\sigma(S_j) = S_i$, for $i \neq j$ and $\sigma(S_h) = S_h$ for all $h \neq i, h \neq j$ (i.e., σ permutes only S_i and S_j), then formula $\beta_{\leq}(S_i, S_j)$ is a candidate symmetry-breaking formula for σ, where $\beta_{\leq}$ is defined as:

$$\beta_{\leq}(S', S'') \doteq |S'| \leq |S''|. \tag{4.18}$$

As another example, consider formula (4.5). It can be generalized as follows, in order to obtain a proper template. Let the UVS σ for specification ψ swap predicates S_i and S_j only. Let us also assume that a linear order '<' exists among elements of the Herbrand domain. Formula $\beta_{least}(S_i, S_j)$ is a candidate symmetry-breaking formula for σ, where β_{least} is defined as:

$$\beta_{least}(S', S'') \doteq \forall X''\ S''(X'')\ \rightarrow\ \exists X'\ S'(X')\ \wedge\ X' < X''. \tag{4.19}$$

It is possible to derive much more complex symmetry-breaking formulae by composing simpler ones. As an example, in Appendix C we show how to model the well-known lex^2 symmetry-breaking constraint (54) for problems that have a "matrix model" with row and column symmetries (e.g., Social golfer or BIBD).

4.6 Experiments

In this section we present some experimental results for assessing the effectiveness of the previously described techniques for some of the problems shown above.

We used state-of-the-art linear and constraint programming solvers, and wrote both a linear and a non linear specification, solved by using Ilog's Cplex and Solver, respectively.

In particular, we focused on the following specifications:

- Graph coloring (we actually focused on the k-coloring version) on some of the benchmark instances already examined in Section 2.4;
- Not-all-equal Sat, on random instances in 3-CNF;
- The Social golfer problem, on some instances, up to 6 players.

The symmetry-breaking formulae we added to the Graph k-coloring specification are the generalizations of (4.5) (least index coloring) and (4.7) (cardinality-based ordering) to the case of k colors. In fact, using the generalization of (4.8) resulted in poorer performances: intuitively, solutions of coloring problems are rarely "color unbalanced", therefore adding too many constraints may not be effective. As shown in Table 4.1, adding our symmetry-breaking formulae (in particular the generalization of (4.7)) seems to be effective especially when Cplex is used. Solver seems to be often negatively affected (hence, detailed results have been omitted), even if with some, but significant, exceptions, and in some cases it is outperformed by Cplex. The same behavior has been observed for Not-all-equal Sat (we solved few hundreds random instances in 3-CNF).

For what concerns the Social golfer problem, we implemented the lex^2 symmetry-breaking formula for the non-linear specification. Differently from the other examples, on this problem Solver is affected positively on several instances, especially negative ones, as Table 4.2 shows.

It is worth noting that both Solver and Cplex have built-in features for symmetry cuts, that can be safely used because of Proposition 4.4.1. All experiments have been done both enabling and disabling such a feature (times in Table 4.1 are with no symmetry cuts). In general, effectiveness of our technique seems independent on using that feature; the built-in feature results anyway in further speed-up when used along with our formulae (thus confirming the intuition given in Chapter 1 on the possibility to apply both specification and instance level optimization techniques).

4.7 Discussion and future research directions

In this chapter we focused on the problem of detecting and breaking symmetries in constraint problem specifications, and presented formal definitions of uniform value transformation (UVT) and symmetry (UVS) in this context, lifting those for CSPs. Furthermore, we gave a logical characterization of UVSs, and showed how the problem of checking whether a UVT is a symmetry for a given specification reduces to checking semantic properties of first-order formulae. Once symmetries have been detected, several approaches can in general be used in order to exploit them. We focused on adding new constraints to the problem specification in order to break detected symmetries. To this end, we defined what is a symmetry-breaking formula, and showed that also the problem of checking whether a given formula is symmetry-breaking for a specification with respect to a given symmetry reduces to checking semantic properties of logical formulae. We detected and broken symmetries on several problem specifications, e.g., Graph 3-coloring, Not-all-equal Sat, Social golfer, and experimental results show that this kind of reasoning can be much effective for the efficient solution of constraint problems, on different solvers.

It is worth noting that the two proposed approaches for symmetry detection and breaking are conceptually independent on each other. In particular, the detection of symmetries of the problem at hand is a necessary prerequisite also for alternative techniques for symmetry breaking, e.g., those that break symmetries during search (cf. Section 4.1).

In this chapter we addressed exclusively uniform value symmetries, but the same approach can be extended to other kind of transformations (cf. Section 4.1). As for future work, we plan to investigate such extensions. As an example, Figure 4.1 shows an opl specification for the N-queens problem, cf. (124, Section 2.2, Statement 2.16), which states that three constraints must hold for all pairs of distinct rows (cf. the con-

			Cplex				
			No s.b.	β_{least}		$\beta_{\leq}$	
Instance	k	Sol?	Time	Time	% sav.	Time	% sav.
DSJC1000.1	24	N	–	368.46	>89.77	–	–
DSJC125.5	8	N	15.32	13.21	13.77	10.51	31.40
DSJC125.5	25	Y	–	2337.29	>35.08	2177.21	39.52
DSJC125.9	21	N	1408.23	2080.21	-47.72	1088.65	22.69
DSJC250.5	10	N	–	2158.75	>40.03	2432.55	32.43
DSJC500.1	11	N	2.53	–	$-\infty$	–	$-\infty$
fpsol2.i.2	21	N	139.80	43.70	68.70	102.20	26.90
fpsol2.i.2	31	Y	–	397.61	>88.96	–	–
fpsol2.i.3	31	Y	–	330.22	>90.83	–	–
le450_25a	21	N	84.73	95.32	-12.50	46.51	45.11
le450_25a	25	Y	3536.41	–	<-1.80	1783.23	49.58
miles500	19	N	2.31	–	$-\infty$	1.67	27.71
mulsol.i.1	30	N	–	10.61	>99.71	–	–
mulsol.i.1	49	Y	–	311.12	>91.36	–	–
mulsol.i.2	30	N	–	10.98	>99.70	–	–
mulsol.i.2	31	Y	26.75	48.67	-81.94	–	$-\infty$
mulsol.i.3	30	N	–	10.78	>99.70	–	–
mulsol.i.3	31	Y	55.77	43.65	21.73	284.32	-409.81
mulsol.i.4	30	N	–	10.99	>99.69	–	–
mulsol.i.4	31	Y	47.46	14.25	69.97	–	$-\infty$
mulsol.i.5	30	N	–	11.12	>99.69	–	–
mulsol.i.5	31	Y	166.85	20.68	87.61	64.56	61.31
myciel4	4	N	5.22	0.87	83.33	8.46	-62.07

Table 4.1: Solving times (seconds) for k-coloring ('–' means that the solver did not terminate in one hour).

Instance				Solver		
Players	Weeks	Groups	Solvable?	No s.b.	β_{lex^2}	% sav.
6	6	3	N	2267.3	2.1	+99.9
6	7	3	N	273.5	4.2	+98.5
6	8	3	N	96.6	10.3	+89.3
9	5	3	N	–	1.0	+99.9
9	6	3	N	342.2	3.8	+98.9

Table 4.2: Solving times (seconds) for Social golfer ('–' means that the solver did not terminate in one hour).

```
int+ N = ...;
range Row 1..N;  range Col 1..N;
var Col Queen[Row];
solve {
   forall (r1, r2 in Row : r1 <> r2) {
      Queen[r1] <> Queen[r2];           // no vertical attack
      Queen[r1] + r1 <> Queen[r2] + r2;  // no NW-SE diagonal attack
      Queen[r1] - r1 <> Queen[r2] - r2;  // no NE-SW diagonal attack
}};
```

Figure 4.1: OPL specification for the N-queens problem.

dition `r1 <> r2`). For symmetry reasons, a solution-preserving (and possibly more efficient) formulation requires the constraints to hold just for *totally ordered* pairs of rows, i.e., `r1 < r2`. From a CSP point of view, this is an example of symmetries on *variables* (94), and it can be recognized simply by proving that swapping `r1` and `r2` leads to an equivalent specification. In a ESO specification, this symmetry can be checked by swapping universally quantified first-order variables, and proving that the equivalence holds. We plan to give formal and general definitions for this and other classes of symmetries, by extending those already given in the CSP literature, and perform a more exhaustive experimentation in order to check whether such generalizations are effective in practice.

As for symmetry-breaking, this chapter suggests, in principle, a guess and test approach, via the use of a library of templates. Hence, an important direction for future work is to address the task of automatically synthesize guaranteed-correct symmetry-breaking formulae for detected symmetries.

Results in this chapter have been published in (19).

Chapter 5

Detecting and exploiting functional dependencies

5.1 Introduction

In this chapter we exploit another interesting property of constraint problems, i.e., the functional dependencies that can hold among predicates in problem specifications. Informally, given a problem specification of the kind (1.1), a guessed predicate S in $\vec{S}$ is said to be functional dependent on the others if, for every solution of any instance, its extension is determined by the extensions of the others.

The presence of functional dependencies is very common in declarative problem specifications for different reasons: as an example, to allow the user to have multiple views of the search space, in order to be able to express the various constraints under the most convenient viewpoint, or to maintain aggregate or intermediate results needed by some of the constraints, as the following example shows.

Example 5.1.1 (The HP 2D-Protein folding problem (84)). *This problem specification models a simplified version of one of the most important problems in computational biology. It consists in finding the spatial conformation of a protein (i.e., a sequence of amino-acids) with minimal energy.*

The simplifications with respect to the real problem are twofold: firstly, the 20-letter alphabet of amino-acids is reduced to a two-letter alphabet, namely H and P. H represents hydrophobic *amino-acids, whereas P represents polar or* hydrophilic *amino-acids. Secondly, the conformation of the protein is limited to a bi-dimensional discrete space. Nonetheless, these limitations have been proven to be very useful for attacking the whole protein conformation prediction protein, that is known to be NP-complete (41) and very hard to solve in practice.*

In this formulation, given the sequence (of length n) of amino-acids of the protein (the so called primary structure of the protein), i.e., a sequence of length n with elements in $\{H,P\}$, we aim to find a connected shape of this sequence on a bi-dimensional grid (whose points have coordinates in the integral range `Coord` $= [-(n-1),(n-1)]$, *the sequence starting at $(0,0)$), which is non-crossing, and maximizes the number of "contacts", i.e., the number of non-sequential pairs of H amino-acids for which the*

Euclidean distance of the positions is 1 (the overall energy of the protein is defined as the opposite of the number of contacts).

Different alternatives for the search space obviously exist: as an example, we can guess the position on the grid of each amino-acid in the sequence, and then force the obtained shape to be connected, non-crossing, and with minimal energy. A second approach is to guess the shape of the protein as a connected path starting at $(0, 0)$, by guessing, for each position t of the sequence, the direction that the amino-acid at the t-th position in the sequence assumes with respect to the previous one (directions in the HP-2D model can only be North, South, East, West). It is easy to show that the former model would lead to a search space of $(2n)^{2n}$ points, while the latter to a much smaller one (4^n points).[1]

However, choosing the latter model is not completely satisfactory. In fact, to express the non-crossing constraint, and to compute the number of contacts in the objective function, absolute coordinates of each amino-acid in the sequence must be computed and maintained. It is easy to show that these values are completely defined by (i.e., functionally dependent on) the sequence of directions taken by the protein. □

Given a problem like the HP 2D-Protein folding one, if writing a procedural program, e.g., in C++, to solve it, possibly using available libraries for constraint programming, a smart programmer would avoid predicates encoding absolute coordinates of amino-acids to be part of the search space. Extensions for these predicates instead, would be *computed* starting from extensions of the others.

On the other hand, when using a declarative language for constraint modelling, the user looses the power to distinguish among predicates whose extension has to be found through a true search, from those which can be computed starting from the others, since all of them become of the same nature. Hence, the search space actually explored by the system can be ineffectively much larger, and additional information should be required from the user to distinguish among them, thus greatly reducing the declarativeness of the specification. To this end, the ability of the system to *automatically recognize* whether a predicate is functionally dependent on (or defined from) the others becomes of great importance from an efficiency point of view, since it can lead to significant reductions of the search space, although retaining the highest level of declarativeness.

The technique of avoiding branches on dependent predicates has already been successfully applied at the instance level for solving, e.g., SAT instances. As an example, it is shown in (65) how to modify the Davis-Putnam procedure for SAT so that it avoids branches on variables added during the clausification of non-CNF formulae, since values assigned to these variables depend on assignments to the other ones. Moreover, some SAT solvers, e.g., EQSATZ (86), have been developed in order to appropriately handle (by means of the so-called "equivalence reasoning") equivalence clauses, that have been recognized to be a very common structure in the SAT encoding of many hard real-world problems, and a major obstacle to the Davis-Putnam procedure.

We believe that looking for dependent predicates at the specification level, rather than after instantiation, can be much more natural, since these issues strongly depend

[1]Actually, as for the second model, possible directions of each amino-acid with respect to the previous one can be only three, because of the non-crossing constraint. Nonetheless, we opt for the simpler model to enhance readability.

on the structure of the problem. To this end, our approach is to give a formal characterization of functional dependencies suitable to be checked by computer tools, and ultimately, to transform the original problem specification by automatically adding an explicit search strategy that exploits such dependencies, avoiding branches on dependent predicates. Moreover, in those cases in which functional dependencies derive from the adoption of multiple viewpoints of the search space, we could choose the best maximal set of independent predicates to branch on, depending on the amenability of the relevant constraints to propagation (cf., e.g, (71)).

The outline of the chapter is as follows: in Section 5.2 we formally define functional dependencies in problem specifications, and characterize them in terms of semantic properties of first-order formulae. This characterization can then be used, in practical circumstances, to exploit computer tools for recognizing functional dependencies (cf. forthcoming Chapter 6). Then, in Section 5.3 we show some examples (using the well-known constraint modelling language OPL) from bio-informatics, planning and resource allocation fields, that exhibit dependencies. In Section 5.4 we describe our approach for exploiting detected dependencies, that consists in the automatic synthesis of a search procedure that explicitly avoids branches on dependent predicates, and show how, in many cases, OPL benefits from this addition. Finally, Section 5.6 is devoted to discussion and description of future work.

5.2 Definitions and formal results

In this section we give the formal definition of functionally dependent guessed predicate in a specification, and show how the problem of checking whether a set of guessed predicates is dependent on the others reduces to check semantic properties of a first-order formula.

Definition 5.2.1 (Functional dependence of a set of predicates in a specification). *Given a problem specification $\psi \doteq \exists \vec{S}\vec{P}\ \phi(\vec{S}, \vec{P}, \vec{R})$ (in which the set of guessed predicates is partitioned into $\vec{S}$ and $\vec{P}$) with input schema $\vec{R}$, $\vec{P}$* functionally depends *on $\vec{S}$ if, for each instance $\mathcal{I}$ of $\vec{R}$ and for each pair of interpretations M, N of $(\vec{S}, \vec{P})$ it holds that, if*

1. *$M \neq N$, and*
2. *$M, \mathcal{I} \models \phi$, and*
3. *$N, \mathcal{I} \models \phi$,*

then $M_{|\vec{S}} \neq N_{|\vec{S}}$, where $\cdot_{|\vec{S}}$ denotes the restriction of an interpretation to predicates in $\vec{S}$.

The above definition states that $\vec{P}$ functionally depends on $\vec{S}$, or that $\vec{S}$ functionally determines $\vec{P}$, if it is the case that, regardless of the instance, each pair of distinct solutions of ψ must differ on predicates in $\vec{S}$, which is equivalent to say that no two different solutions of ψ exist that coincide on the extension for predicates in $\vec{S}$ but differ on that for predicates in $\vec{P}$.

Example 5.2.1 (Graph 3-coloring, Example 1.2.2 continued). *In the Graph 3-coloring problem specification, one of the three guessed predicates R, G, B is functionally dependent on the others. As an example, B functionally depends on R and G, since, regardless of the instance, it can be defined as*

$$\forall X \; B(X) \;\leftrightarrow\; \neg(R(X) \vee G(X)).$$

This is because constraint (1.6) *is equivalent to*

$$\forall X \; \neg(R(X) \vee G(X)) \rightarrow B(X)$$

and (1.8) *and* (1.9) *imply*

$$\forall X \; B(X) \;\rightarrow\; \neg(R(X) \vee G(X)).$$

In other words, for every input instance, no two different solutions exist that coincide on the set of red and green nodes, but differ on the set of blue ones.

Example 5.2.2 (Not-all-equal Sat (Example 1.2.3 continued)). *In the Not-all-equal problem specification, one of the two guessed predicates T and F is dependent on the other, since by constraints (1.13–1.14) it follows, e.g., that*

$$\forall X \; F(X) \;\leftrightarrow\; var(X) \wedge \neg T(X).$$

The two examples above are very simple, and have been presented to give just the main intuition. It can be observed how in both cases, the presence of functional dependencies arises because of the non-overlapping constraints among different predicates. However, in the remainder of the chapter, much more complex examples of problem specifications that exhibit functional dependencies will be shown.

It is worth noting that Definition 5.2.1 is strictly related to the concept of *Beth implicit definability*, well-known in logic (cf., e.g., (31)). We will further discuss this relationship in Section 5.4.

In what follows, instead, we show that the problem of checking whether a subset of the guessed predicates in a specification is functionally dependent on the remaining ones, reduces to semantic properties of a first-order formula (proofs are delayed to Appendix D). To simplify notations, given a list of predicates $\vec{T}$, we write $\vec{T'}$ for representing a list of predicates of the same size with, respectively, the same arities, that are fresh, i.e., do not occur elsewhere in the context at hand. Also, $\vec{T} \equiv \vec{T'}$ will be a shorthand for the formula

$$\bigwedge_{T \in \vec{T}} \forall \vec{X} \; T(\vec{X}) \;\leftrightarrow\; T'(\vec{X}),$$

where T and T' are corresponding predicates in $\vec{T}$ and $\vec{T'}$, respectively, and $\vec{X}$ is a list of variables of the appropriate arity.

Theorem 5.2.1. *Let $\psi \doteq \exists \vec{S}\vec{P} \; \phi(\vec{S}, \vec{P}, \vec{R})$ be a problem specification with input schema $\vec{R}$. $\vec{P}$ functionally depends on $\vec{S}$ if and only if the following formula is valid:*

$$[\phi(\vec{S}, \vec{P}, \vec{R}) \wedge \phi(\vec{S'}, \vec{P'}, \vec{R}) \wedge \neg(\vec{S}\vec{P} \equiv \vec{S'}\vec{P'})] \rightarrow \neg(\vec{S} \equiv \vec{S'}). \qquad (5.1)$$

Unfortunately, the problem of checking whether the set of predicates in $\vec{P}$ is functionally dependent on the set $\vec{S}$ is undecidable, as the following result shows:

Theorem 5.2.2. *Given a specification on input schema $\vec{R}$, and a partition $(\vec{S}, \vec{P})$ of its guessed predicates, the problem of checking whether $\vec{P}$ functionally depends on $\vec{S}$ is not decidable.*

Even if Theorem 5.2.2 states that checking functional dependencies in a specification is an undecidable task, in practical circumstances it can be effectively and often efficiently performed by automatic tools, such as first-order theorem provers and finite model finders (cf. Chapter 6). Hence, in order to automatically recognize functional dependencies in a problem specification, an approach similar to that of Section 4.3 for detecting uniform value symmetries can, in principle, be used. Of course, the same observations regarding the number of checks to be made in practical circumstances hold also in this case.

5.3 Further examples

In this section we present some problem specifications that exhibit functional dependencies among guessed predicates. Due to their high complexity, we don't give their formulations as ESO formulae, but show their specifications in the well known language for constraint modelling OPL.

Example 5.3.1 (The HP 2D-Protein folding problem, Example 5.1.1 continued). *As already stated in Example 5.1.1, rather than guessing the position on the grid of each amino-acid in the sequence, we chose to represent the shape of the protein by a guessed predicate* `Moves[]`*, that encodes, for each position t, the direction that the amino-acid at the t-th position in the sequence assumes with respect to the previous one (the sequence starts at $(0,0)$). An* OPL *specification for this problem is shown in Appendix H.1.1. The analogous of the instance relational schema, guessed predicates, and constraints can be clearly distinguished in the* OPL *code.*

To express the non-crossing constraint, and to compute the number of contacts in the objective function (not shown in the OPL *code for brevity), absolute coordinates of each amino-acid in the sequence must be calculated (predicates* `X[]` *and* `Y[]`*). The non-crossing constraint can be expressed by stating $\mathcal{O}(n^2)$ binary inequalities, i.e., that do not exist two different elements t and t' for which* `X[t] = X[t']` *and* `Y[t] = Y[t']`*. From the considerations above, it follows that guessed predicates* `X[]` *and* `Y[]` *are functionally dependent on* `Moves[]`*.*

To test the effectiveness of the approach, we considered also a second specification for HP 2D-Protein folding, where the non-crossing constraint is modelled differently. In particular, a new guessed predicate `Hits[]` *is used, in order to maintain, for every position on the grid, the number of amino-acids of the protein that are placed on it at each point during the construction of the shape. For all of them, this number cannot be greater than 1, that implies that the string does not cross. In this specification, which* OPL *code is presented in Appendix H.1.2, also guessed predicate* `Hits[]` *is defined by* `Moves[]`*.*

Example 5.3.2 (The Sailco inventory problem (124, Section 9.4, Statement 9.17)). *This problem specification, part of the* OPLSTUDIO *distribution package (as file* `sailco.mod`*), models a simple inventory application, where the question is to decide how many sailboats the Sailco company has to produce over a given number of time periods, in order to satisfy the demand and to minimize production costs. The demand for the periods is known and, in addition, an* `inventory` *of boats is available initially. In each period, Sailco can produce a maximum number of boats (*`capacity`*) at a given unitary cost (*`regularCost`*). Additional boats can be produced, but at higher cost (*`extraCost`*). Storing boats in the inventory also has a cost per period (*`inventoryCost` *per boat).*

Section A.2 shows an OPL *model for this problem. From the specification it can be observed that the amount of boats in the inventory for each time period* `t` > 0 *(i.e.,* `inv[t]`*) is defined in terms of the amount of regular and extra boats produced in period* `t` *by the following relationship:* `inv[t] = regulBoat[t] + extraBoat[t] - demand[t] + inv[t-1]`.

Example 5.3.3 (The Blocks world problem (102; 130)). *In the Blocks world problem, the input consists of a set of blocks that are arranged in stacks on a table. Every block can be either on the table or onto another block. Given the initial and the desired configurations of the blocks, the problem amounts to find a minimal sequence of moves that allows to achieve the desired configuration. Every move is performed on a single clear block (i.e., on a block with no blocks on it) and moves it either onto a clear block or on the table (that can accommodate an arbitrary number of blocks). It is worth noting that a plan of length less than or equal to twice the number of blocks always exists, since original stacks can all be flattened on the table before building the desired configuration.*

In our formulation, given in Appendix H.3, we assume that the input is given as an integer `nblocks`*, i.e., the number of blocks, and functions* `OnAtStart[]` *and* `OnAtGoal[]`*, encoding, respectively, the initial and desired configuration. As for the guessed functions,* `MoveBlock[]` *and* `MoveTo[]` *respectively state, for each time point* `t`*, which block has been moved at time point* `t-1`*, and its new position at time* `t`*. Moreover, we use guessed functions* `On[]`*, that states the position (which can be either a block or the table) of a given block at a given time point, and* `Clear[]`*, that states whether a given block is clear at a time point. We observe that guessed functions* `On[]` *and* `Clear[]` *are functionally dependent on* `MoveBlock[]` *and* `MoveTo[]`. □

5.4 Exploiting functional dependencies

Once a set $\vec{P}$ of guessed predicates of a problem specification $\psi \doteq \exists \vec{S}\vec{P}\ \phi(\vec{S}, \vec{P}, \vec{R})$ has been recognized to be functionally dependent on the others, different approaches can be followed in order to exploit such a dependence: the most simple and elegant one is to force the system to branch only on defining predicates, i.e., those in $\vec{S}$, avoiding spending search on those in $\vec{P}$. An alternative approach is to substitute in the specification all occurrences of predicates in $\vec{P}$ with their definition, and we will briefly discuss it in the following.

As already observed in Section 5.2, the concept of functional dependence among guessed predicates expressed in Definition 5.2.1 is strictly related to the one of *Beth* *implicit definability* (cf., e.g, (31)). In particular, given a problem specification $\psi \doteq \exists \vec{S}\vec{P}\ \phi(\vec{S}, \vec{P}, \vec{R})$, guessed predicates in set $\vec{P}$ functionally depend on those in $\vec{S}$ if and only if the first-order formula $\phi(\vec{S}, \vec{P}, \vec{R})$ implicitly defines predicates in $\vec{P}$ (i.e., if every $\langle\vec{S}, \vec{R}\rangle$-structure has at most one expansion to a $\langle\vec{S}, \vec{P}, \vec{R}\rangle$-structure satisfying $\phi(\vec{S}, \vec{P}, \vec{R})$).

It is worth remarking that, since we are interested in finite extensions for guessed predicates, Beth implicit definability has to be intended *in the finite*. Now, the question that arises is whether it is possible to derive, once a functional dependence (or, equivalently, an implicit definition) has been established, a formula that *explicitly defines* the dependent predicates in terms of the others. This formula, then, could take the place of all occurrences of those predicates in the problem specification. Although such a formula always exists in unrestricted first-order logic, this is not the case when only finite models are allowed. This is because first-order logic does not have the *Beth property in the finite* (cf., e.g., (48), and the intrinsically inductive definition of guessed function `inv[]` in Example 5.3.2). Nonetheless, in some cases such a formula indeed exists (cf., e.g., Examples 5.2.1 and 5.2.2). On the other hand, a second-order explicit definition of a dependent predicate would not be adequate, since new quantified predicates have to be added to the specification, and, moreover, the obtained specification may not be in ESO any more. Hence, we follow the first approach, i.e., forcing the system to avoid branches on dependent predicates, in order to save search.

In this section we show how an explicit search strategy that avoids branches on dependent predicates can be automatically synthesized. To this end, we assume to face with languages that allow the user to provide an explicit search strategy, and, in particular, since its description may depend on the particular solver, we present examples by focusing on the constraint language OPL.

OPL does not require a search strategy (called, in the OPL syntax, "search procedure") to be defined by the user, since it automatically uses default strategies when none is explicitly defined. On the other hand, it provides the designer with the possibility of explicitly programming in detail how to branch on variables, and how to split domains, by adding a `search` part in the problem specification.

The simplest way to provide a search procedure is by using the `generate()` construct, which receives a guessed predicate as input and forces OPL to generate all possible extensions for it, letting the policy for the generation to the system defaults. Of course, multiple occurrences of `generate()` (with different guessed predicates as arguments) are allowed. OPL also allows more explicit search procedures, where the programmer can explicitly choose the order for the generation of the various extensions for that predicate (cf. (124)). However, these topics require an accurate knowledge of the particular problem at hand, and are out of the scope of this chapter, since they are less amenable to be generalized and automated.

Hence, given a problem specification in which a set $\vec{P}$ of the guessed predicates is functionally dependent on the others (set $\vec{S}$), a search procedure that forces OPL to avoid branches on predicates in $\vec{P}$ is the following:

```
search {
  generate(S1);  ...  generate(Sm);
  generate(P1);  ...  generate(Pk);
};
```

where $\{S1, \ldots, Sm\}$ is a suitable permutation of the m predicates in $\vec{S}$, and $\{P1, \ldots, Pk\}$ is a suitable permutation of the m predicates in $\vec{P}$ (the actual permutations chosen may affect only efficiency). This search strategy does not completely exclude branches on dependent predicates, but it makes such branches only after a generation of the extension for all defining predicates has already been made (and its effects propagated). Removing the generation of extensions for dependent predicates may, in general, invalidate completeness. However, in case the computation of the extensions for predicates in $\vec{P}$ can be made by using only constraint propagation (which arises frequently in practice, cf., e.g., all the examples above), calls to `generate(Pi)` for `i` $\in [1, k]$ do not actually produce non-determinism, since the domains for the relevant variables have already been reduced to singletons, and hence can be omitted. Anyway, in these cases, their inclusion is not expected to produce a significant overhead.

Search procedures added to the OPL specifications of HP 2D-Protein folding, Sailco inventory, and Blocks world problems in order to deal with the above described dependencies are, respectively, the following ones:

```
search {
  generate(Moves);
  generate(X);
  generate(Y);
};

search {
  generate(regulBoats);
  generate(extraBoats);
  generate(inv);
};

search {
  generate(MoveBlock);
  generate(MoveTo);
  generate(On);
  generate(Clear);
};
```

It is worth noting that, for some specifications, sets $\vec{S}$ and $\vec{P}$ are interchangeable. This intuitively happens when the user adopts multiple viewpoints of the search space (cf., e.g., Example 5.3.1, in which set $\{X, Y\}$ depends on $\{Moves\}$ and vice versa). In those cases, a first choice for deciding which set should be regarded as "defining" (i.e., $\vec{S}$), may involve the size of the associated search space, but other and smarter approaches can be used (cf., e.g., (71)).

5.5 Experiments

To test the effectiveness of adding the automatically synthesized search procedures presented in Section 5.4, we made an experimentation with OPL. In particular, we made the following experiments:

- The HP 2D-Protein folding problem on benchmark instances, some of them taken from (70);
- The Blocks world problem, on structured instances, some of them used as benchmarks in (78);
- The Sailco inventory problem on random instances.

As for HP 2D-Protein folding, we used both specifications described in Example 5.3.1. Adding a search procedure that explicitly avoids branches on dependent predicates always speeds-up the computation, and often the saving in time in quite consistent. Table 5.1 reports typical behaviors of the system (specification denoted by “BI” is the one with binary inequalities for non-crossing, cf. Appendix H.1.1, while that denoted by “Hits” is the one with additional guessed predicate Hits, cf. Appendix H.1.2). It is worth noting that benefits are very impressive for what concerns the specification “Hits”. This is an example of how automated reasoning on problem specifications can recover some inaccuracies made by designers, when writing “low quality” models.

Also results for the Blocks world problem show that avoiding branches on dependent predicates greatly speed-ups the computation. Time savings are often impressive in this case, as Table 5.2 shows.

On the other hand, for what concerns the Sailco inventory problem, no saving in time has been observed. This can be explained by observing that the problem specification is linear, and the linear solver automatically chosen by the system (i.e., CPLEX) is built on different technologies (e.g., the simplex method) that are not amenable to this kind of program transformation.

5.6 Discussion and future research directions

In this chapter we discussed a semantic logical characterization of functional dependencies among guessed predicates in declarative constraint problem specifications. Functional dependencies can be very easily introduced during declarative modelling, either because intermediate results have to be maintained in order to express some of the constraints, or because of precise choices, e.g., redundant modelling. However, dependencies can negatively affect the efficiency of the solver, since the search space can become much larger, and additional information from the user is today needed in order to efficiently cope with them.

We described how, in our framework, functional dependencies can be checked by computer, and can lead to the automated synthesis of search strategies that avoid the system spending search in unfruitful branches. Several examples of constraint problems that exhibit dependencies have been presented, from bio-informatics, planning, and resource allocation fields, and experimental results have been discussed, showing that current systems for constraint programming greatly benefit from the addition of such search strategies.

As for future work, we plan to investigate the actual complexity of finding the extensions for dependent predicates, once a guessing on defining ones has been made. In particular, our investigations will focus on problems that are guaranteed to have exactly one solution.

Spec.	Length	max. contacts	Time		% saving
			w/out search proc.	with search proc.	
BI	14	5	47.73	39.56	17.12
BI	14	2	33.83	33.44	1.15
BI	16	7	26.91	22.57	16.13
BI	16	6	127.53	113.36	11.11
BI	17	6	2787.64	2136.22	23.37
BI	17	6	316.83	286.37	9.61
BI	18	4	575.36	493.75	14.18
BI	18	?	–	–	–
BI	20	?	–	–	–
Hits	6	1	157.42	0.53	99.66
Hits	6	2	377.72	0.89	99.76
Hits	7	0	–	0.53	>99.99
Hits	7	2	–	0.62	>99.98
Hits	8	1	–	0.46	>99.99
Hits	8	2	–	0.57	>99.98
Hits	8	3	–	2.68	>99.93

Table 5.1: OPL solving times for benchmark instances of the HP 2D-Protein folding problem, with and without search procedure. ('–' means that OPL did not terminate in one hour.)

Results in this chapter have been published in (21; 22).

Instance	Blocks	Minimal plan length	Time		Saving %
			w/out search proc.	with search proc.	
bw-sussman	3	3	12.48	0.42	96.63
bw-reversal4	4	4	1.23	1.16	5.69
bw-4.1.1	4	6	882.72	0.66	99.92
bw-4.1.2	4	5	1879.25	0.87	99.95
bw-5.1.1	5	8	–	1.21	>99.97
bw-5.1.3	5	7	935.32	27.53	97.06
bw-large-a	9	12	–	–	–

Table 5.2: OPL solving times for some benchmark instances of the Blocks world problem, with and without search procedure. ('–' means that OPL did not terminate in one hour.)

Chapter 6

Automated reasoning on problem specifications

6.1 Introduction

In previous chapters we presented different techniques for reformulating constraint problem specifications and showed how the problem of recognizing the relevant properties of specifications can be reduced in checking semantic conditions of first-order formulae. This implies that automated tools can be used, in principle, to automatically make the required reasoning.

This chapter is devoted to this topic. In particular, we discuss the use of first-order theorem provers and finite model finders for checking properties of problem specifications suitable for being optimized. The main result is that, even if the underlying problems have been proven to be undecidable (cf. Theorems 4.3.2 and 5.2.2), in practical circumstances such tools perform well on these reasoning tasks.

Relations between constraint satisfaction and deduction have been observed since several years (cf., e.g., the early work (8), and (81) for an up-to-date report). Indeed, not much work has been done on reasoning at the specification level (the only example, to the best of our knowledge, is CGRASS (60), that can be potentially used to automatically infer properties also on constraint models, or classes of CSPs). On the other hand, by using ESO as the formalism for specifying constraint problems, it becomes natural –in principle– to infer properties of the specifications by means of automated reasoning tools. Hence, the purpose of this chapter is exactly to link two important technologies: automated theorem proving (ATP) and constraint programming.

We report the results on using first-order theorem provers and finite model finders when focusing on two forms of reasoning:

- Checking existence of *value symmetries*, i.e., properties of the specification that permit to exchange values of the finite domains without losing all solutions (cf. Chapter 4); on top of that, we check whether a given formula breaks such symmetries or not;
- Checking existence of *functional dependencies*, i.e., properties that force values of some guessed predicates to depend on the value of other ones (cf. Chapter 5).

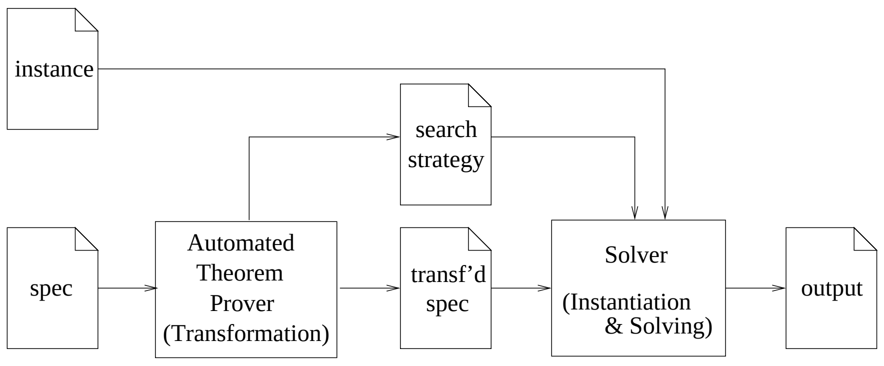
instance
search strategy
spec
Automated Theorem Prover (Transformation)
transf'd spec
Solver (Instantiation & Solving)
output

Figure 6.1: Architecture of the problem solving system.

There are at least two reasons why a system should make automatically such checks: in Chapters 4 and 5 we already showed how exploiting of, respectively, symmetries and functional dependencies, can make the solving process more efficient, and provided several examples of specifications that exhibit these properties.

On the other hand, the automated detection of such properties is needed also for verification purposes. In fact, the person performing constraint modelling may be interested in the above properties: as an example, existence (or lack) of a dependence may reveal a bug in the specification.

The architecture of the system we envision is represented in Figure 6.1. Hence, the output of the transformation phase is a reformulated constraint program and, possibly, a search strategy.

In what follows we show that it is actually possible to use ATP technology to reason on combinatorial problems and to reformulate them. As a side-effect, we propose a new domain of application and a brand new set of benchmarks for ATP systems, that is not represented in large repositories such as TPTP[1].

Sections 6.2 and 6.3 are devoted to the description of experiments in checking symmetries and dependencies, respectively, while Section 6.4 draws conclusions and presents current research. Appendixes H and E show some files for, respectively, OPL and the ATP used.

6.2 Detecting and breaking uniform value symmetries

In this section we face the problem of automatically detecting and breaking value symmetries in problem specifications. To this end, we refer to definitions and results in Chapter 4, and in particular to the concepts of Uniform value transformation (UVT) and symmetry (UVS) (cf. Definitions 4.2.4 and 4.2.5, respectively), to Theorem 4.3.1 and to the definition of symmetry-breaking formula (cf. Definition 4.4.1).

Given an ESO problem specification $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$, Theorem 4.3.1 and Definition 4.4.1 claim that, checking whether a given UVT σ is a symmetry for ψ, or whether

[1]`http://www.tptp.org`.

a formula $\beta(\vec{S})$ is breaks a given symmetry, reduces to check semantic properties of first-order formulae.

In the following, we give some details about the experimentation done using automated tools. First of all we note that, obviously, all the above mentioned conditions can be checked by using a refutation *theorem prover*. It is interesting to note that, for some of them, we can use a *finite model finder*. In particular, we can use such a tool for checking statements (such as condition (4.1) of Definition 4.4.1 or the negation of the condition of Proposition 4.3.1) that are syntactically a non-equivalence. As a matter of facts, it is enough to look for a finite model of the negation of the statement, i.e., the equivalence. If we find such a model, then we are sure that the non-equivalence holds, and we are done. The tools we used are the first-order refutation theorem prover OTTER (93) and the first-order model finder MACE (92). We invoked both tools in full "automatic" mode.

6.2.1 Detecting symmetries

The examples we worked on are those of Chapter 4, and in particular:

- Graph 3-coloring (Example 4.3.1), with the symmetry $\sigma^{G,B}$ defined as $\sigma^{G,B}(R) = R$, $\sigma^{G,B}(G) = B$, $\sigma^{G,B}(B) = G$;
- Not-all-equal Sat (Example 4.5.2), with the symmetry $\sigma^{T,F}$, defined as $\sigma^{T,F}(T) = F$ and $\sigma^{T,F}(F) = T$;
- Social golfer (Example 4.5.3) in the unfolded version, with symmetries $\sigma_W^{G,G'}$, for all weeks W, swapping $PLAY_{W,G}$ and $PLAY_{W,G'}$.

The results we obtained with OTTER are shown in Table 6.1. The third row refers to the version of the Not-all-equal Sat problem in which all clauses have three literals, the input is encoded using a ternary relation $clause(\cdot,\cdot,\cdot)$, and the specification varies accordingly. It is interesting to see that the performance is good in many cases (cf. Table 6.1). As for the fourth row, it refers to the unfolded specification of the Social golfer problem with 2 weeks and 2 groups of size 2. Unfortunately, OTTER does not terminate for a larger number of weeks or groups (cf. Section 6.4).

A note on the encoding is in order. Initially, we gave the input to OTTER exactly in the format specified by Theorem 4.3.1, but the performance was quite poor: for Graph 3-coloring the tool did not even succeed in transforming the formula in clausal form, and symmetry was proven only for very simplified versions of the problem, e.g., 2-coloring, omitting constraint (1.6). Results of Table 6.1 have been obtained by introducing new propositional variables defining single constraints: as an example, constraint (1.6) is represented as

```
covRGB <-> (all x (R(x) | G(x) | B(x))).,
```

where `covRGB` is a fresh propositional variable.

Obviously, we wrote a first-order logic formula encoding condition of Theorem 4.3.1, and gave its negation to OTTER in order to find a refutation. As an

Spec	Symmetry	CPU time (sec)
3-coloring	$\sigma^{G,B}$	0.27
Not-all-equal Sat	$\sigma^{T,F}$	0.22
Not-all-equal 3-Sat	$\sigma^{T,F}$	4.71
Social golfer (unfolded)	$\sigma_W^{G,G'}$	0.96

Table 6.1: Performance of OTTER for proving that a UVT is a UVS.

example, Appendix E.2.1 shows the whole OTTER code used to check that $\sigma^{R,G}$ is a UVS for the 3-coloring specification.

As for proving non-existence of symmetries, we used the Graph 3-coloring with red self-loops problem of Example 2.2.1, and the UVT $\sigma^{R,G}$ that is not a symmetry for it. We wrote a first-order logic formula encoding condition of Theorem 4.3.1 for $\sigma^{R,G}$ and gave its negation to MACE in order to find a model of the non-equivalence. MACE was able to find the model described in Example 2.2.1 in less than one second of CPU time.

6.2.2 Breaking symmetries

We worked on Graph 3-coloring (Example 4.3.1) with the UVS $\sigma^{G,B}$, and Social golfer (Example 4.5.3) in the unfolded version, with symmetries $\sigma_W^{G,G'}$. As for the candidate symmetry-breaking formulae, we considered some of those described in Examples 4.5.1 and 4.5.3. In particular, we checked both conditions of Definition 4.4.1 with $\beta_{sel}^{G,B}(R,G,B)$ (4.4), $\beta_{least}^{G,B}(R,G,B)$ (4.5), and $\beta_{\leq}^{G,B}(R,G,B)$ (4.6) with respect to $\sigma^{G,B}$ in the Graph 3-coloring specification, and with $\beta_{least\ W}^{G,G'}(PLAY_{W,G}, PLAY_{W,G'})$ (4.16) with respect to $\sigma_W^{G,G'}$ in the Social golfer specification.

As for $\beta_{\leq}^{G,B}(R,G,B)$, some considerations are in order, since this example highlights some difficulties that can arise when using first-order ATPs. In fact, although formula (4.6) can be written in ESO using standard techniques (i.e., an ESO formula that evaluates to true if and only if a total injective function from tuples in R to those in G exists, cf. Appendix A), it is not first-order definable. Therefore, conditions in Definition 4.4.1 are second-order (non-)equivalences, and the use of a first-order theorem prover may not suffice.

However, in some circumstances, it is possible to synthesize first-order conditions that can be used to infer the truth value of those of Definition 4.4.1. In Appendix E.1 we show an example of first-order definable steps that imply that condition (4.1), i.e., point 1 of Definition 4.4.1 holds, for the case of a formula $\beta(\vec{S})$ expressed in ESO, as is, e.g., $\beta_{\leq}^{G,B}(R,G,B)$.

We used MACE and OTTER in order to prove that the formulae described above were symmetry-breaking for the 3-coloring problem specification with respect to $\sigma^{G,B}$, by checking conditions of Definition 4.4.1. As Table 6.2 shows, times are always good when checking the first condition, while may become worse when OTTER is used to prove the second one. On the other hand, we strongly believe that the

Spec	Symmetry	$\beta(\vec{S})$	CPU time (sec)	
			Cond. (4.1) (Mace)	Cond. (4.2) (Otter)
3-coloring	$\sigma^{G,B}$	$\beta_{sel}^{G,B}(R,G,B)$	1.47	1.89
	$\sigma^{G,B}$	$\beta_{least}^{G,B}(R,G,B)$	2.67	?
	$\sigma^{G,B}$	$\beta_{<}^{G,B}(R,G,B)$	0.25	–
Social golfer	$\sigma_{W}^{G,G'}$	$\beta_{least\ W}^{G,G'}(\cdots)$	0.04	?

Table 6.2: Performance of Otter for proving that a formula $\beta(\vec{S})$ is symmetry-breaking for a given symmetry.

poor performances of Otter when dealing with more complex symmetry-breaking formulae can be explained by observing that these tools are strongly optimized for the kind of formulae of benchmark libraries, e.g., TPTP and various competitions (e.g., CASC[2]), that have a structure very different with respect to ours. As a side-effect, we think that the wide availability of problem specifications can offer a brand new set of benchmarks for this kind of tools.

6.3 Detecting functional dependencies

In this section we tackle the problem of the automated recognition of guessed predicates that functionally depend on other ones in a given specification.

In Chapter 5 we have already shown how functional dependencies arise frequently in problem specifications, and that their recognition can positively affect the efficiency of backtracking solvers, leading to the automatic synthesis of search strategies exploiting them, e.g., by avoiding branches regarding dependent predicates. In the same chapter we gave several examples of problem specifications that exhibit functional dependencies, and presented experimental results showing the effectiveness of the approach.

In this section, instead, we show how ATP technology can be effectively used to automate the reasoning described in Chapter 5. In the following, we refer to that chapter for definitions and results: in particular, in this section we make use of Definition 5.2.1 and Theorems 5.2.1 and 5.2.2.

Using Theorem 5.2.1 it is easy to write a first-order formula that is valid if and only if a given dependence holds. We used Otter for proving the existence of dependencies among guessed predicates of different problem specifications:

- *Graph 3-coloring* (cf. Example 5.2.1), where one among the guessed predicates R, G, B is dependent on the others;
- *Not-all-equal Sat* (cf. Example 5.2.2), where one between the guessed predicates T and F is dependent on the other.

For each of the above specifications, we wrote a first-order logic encoding of formula (5.1), and gave its negation to Otter in order to find a refutation.

[2]`http://www.cs.miami.edu/~tptp/CASC/`

For the purpose of testing effectiveness of the proposed technique in the context of more complex specifications written in implemented languages, we considered also:

- *Sailco inventory* problem (cf. Example 5.3.2), where `inv[]` depends on `regulBoat[]` and `extraBoat[]`;
- *HP 2D-Protein folding* problem (cf. Example 5.3.1), where `X[]` and `Y[]` depend on `Moves[]`.

Otter encodings of such problems are given in Appendix E.2.2.

In order to make Otter able to correctly handle expressions of interest (in particular, arithmetic constraints), additional constraints had to be included in the Otter formulae.

As for Sailco (cf. Appendix E.2.2), we opted for an encoding that uses function symbols: as an example, the `inv[]` array (cf. the opl specification in Appendix H.2) is translated to a function symbol $inv(\cdot)$ rather than to a binary predicate. More precisely, according to Theorem 5.2.1, a pair of function symbols $inv(\cdot)$ and $inv'(\cdot)$ is introduced. The same happens for `regulBoat[]` and `extraBoat[]`. Moreover, we included the following additional constraints:

```
equalDiscrete <-> (inv(0) = inv_prime(0) &
                  (all t (t > 0 -> (inv(t) - inv(t-1)) =
                                   (inv_prime(t) - inv_prime(t-1))))).
induction <-> (equalDiscrete -> (all t (inv(t) = inv_prime(t)))).
```

in order to allow Otter to infer $\forall t\ inv(t) = inv'(t)$ from equality of inv and inv' at the initial time period and equivalence of increments in all time intervals of length 1. A similar approach has been followed for HP 2D-Protein folding (cf. Appendix E.2.2).

Results of the experiments to check that dependencies for all the aforementioned problems hold are shown in Table 6.3. As it can be observed, for almost all of them the time needed by Otter is very small. An exception is HP 2D-Protein folding, for which Otter needed several minutes, with the specification showed in Appendix H.1.2, and was not able to find a proof when using the specification in Appendix H.1.1. As for the Blocks world problem specification instead, Otter did not return any proof. To this end, we are investigating the use of other theorem provers (cf. Section 6.4).

6.4 Discussion and future research directions

The use of automated tools for preprocessing CSPs has been almost completely limited, to the best of our knowledge, to the instance level. In this chapter we proved that current ATP technology is able to perform significant forms of reasoning on specifications of constraint problems. We focused on two forms of reasoning: symmetry detection and breaking, and functional dependence checking. Reasoning has been done for various problems, including the ESO encodings of graph 3-coloring and Not-all-equal Sat, and the opl encoding of an inventory problem, 2D HP-Protein folding

Spec	$\vec{S}$	$\vec{P}$	CPU time (sec)	Proof length	Proof level
3-coloring	R, G	B	0.25	27	18
Not-all-equal 3-Sat	T	F	0.38	18	14
Sailco	$regulBoat$, $extraBoat$	inv	0.21	29	11
HP 2D-Prot. folding	$Move$	X, Y	569.55	76	16

Table 6.3: Performance of OTTER for proving that the set $\vec{P}$ of guessed predicates is functionally dependent on the set $\vec{S}$.

and Blocks world problems. In the latter examples, arithmetic constraints exist, and we have shown how they can be handled. In many cases, reasoning is done very efficiently by the ATP, although effectiveness depends on the format of the input, and auxiliary propositional variables seem to be necessary.

There are indeed some tasks, namely, proving existence of symmetries in the *Social golfer problem*, or existence of functional dependencies in the Blocks world problem, which OTTER –in the automatic mode– was unable to do. So far, we used only two tools, namely OTTER and MACE, and plan to investigate effectiveness of other provers, e.g., VAMPIRE (109). We note that the wide availability of constraint problem specifications, both in computer languages, cf., e.g., (57; 124), and in natural language, cf., e.g., (62), the CSP-Library (64), and the OR-Library (104), offers a brand new set of benchmarks for ATP systems, which is not represented in large repositories, such as TPTP.

Currently, the generation of formulae that are given to the theorem prover is performed by hand. Nonetheless, we claim that, as it can be observed from the various examples, this task can in principle be performed automatically, since the OPL syntax for expressing constraints is essentially first-order logic. We plan to investigate this topic in future research, with much attention on how to deal with arithmetic constraints in a general way. In particular, from our experiments, it seems that the complete axiomatizations of integers, linear orders, and arithmetics may in general be avoided, as long as a minimal set of axioms is included in the ATP input formula in order to let the theorem prover able to correctly make the relevant unifications among different arithmetic expressions.

We believe that ATPs can be used also for other useful forms of reasoning, apart from those described in this chapter. As an example, in Figure 4.1 an OPL specification of the N-queens problem is shown. In Section 4.7 we already observed that for symmetry reasons, a solution-preserving (and possibly more efficient) formulation requires the constraints to hold just for *totally ordered* pairs of rows, i.e., `r1 < r2`, and how this kind of symmetries can be detected. OTTER was able to prove that this symmetry on *variables* hold in less than one second of CPU time. As already remarked in Section 4.7, we are currently investigating the applicability of such a technique for a general class of specifications, in which different classes of symmetries hold.

Results in this chapter have been published in (23; 24).

Chapter 7

Recognizing compilable specifications

7.1 Introduction

This chapter deals with a different technique that has been proposed in the Artificial Intelligence literature for addressing polynomial intractability of reasoning: *Knowledge compilation* (KC). The central observation behind KC is that, in many reasoning tasks, the input to a deduction problem $KB \models \delta$ is split into two parts, KB and δ, with different status: Typically KB is not modified very often, and it is used for several instances of the problem. For this reason, KB is sometimes called the *fixed part*, and δ the *varying part*. The idea of KC is to split the deduction problem into two phases (depicted in Figure 7.1):

- In the first one (sometimes called *off-line reasoning*) KB is preprocessed, thus obtaining an appropriate data structure D_{KB};
- In the second phase (sometimes called *on-line reasoning*), the problem $KB \models \delta$ is actually solved using D_{KB} and δ.

The goal of preprocessing is to make on-line reasoning computationally easier (possibly polynomial-time) with respect to reasoning in the case when no preprocessing at all is done. A reasonable requirement is that the size of D_{KB} is polynomial in the size of KB, otherwise it would be unpractical to store the computed information. Intuitively, the effort spent in the preprocessing phase pays off when its computational cost is amortized over the (facilitated) answer to many instances. KC has been analyzed for a variety of logical consequence relations $\models$, including propositional logic (114), belief revision (132), and various non-monotonic formalisms such as default logic (7) and circumscription (101).

In this chapter we study KC in the context of relational databases. In particular, we will assume that both KB and δ are represented in some way as databases, and that logical consequence, i.e., the '$\models$' relation in $KB \models \delta$, is represented as a logical query Q in second-order logic. D_{KB} is also a database, to be *synthesized* from KB

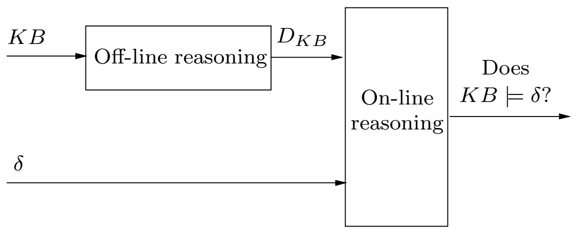

Figure 7.1: Compilation of a problem.

by means of an appropriate *view*. On-line reasoning is also represented as a logical query Q_r, obtained by *rewriting* Q.

We will show syntactic restrictions on the query Q that imply that a polynomial-size D_{KB} and a first-order Q_r exist, thus implying that on-line reasoning can be done in polynomial time, i.e., Q is *compilable* to P. We will also present classes (in some sense complementary to the former ones) of *non-compilable* queries, for which either no polynomial-size D_{KB} or no first-order Q_r exist (unless the Polynomial Hierarchy –PH– collapses).

Focusing on such a restricted case of KC rules out some interesting cases: As an example, since Q is second-order, no problem which is PSPACE-hard can be studied (provided PH $\neq$ PSPACE). Nevertheless, formalizing the deduction problem as a logical formula presents a clear advantage, since we have the possibility to separate compilable and non-compilable queries by means of a syntactic criterion. This helps to understand why "similar" problems may be quite different with respect to compilation.

In fact a vast literature both on the pragmatic and the theoretical aspects of KC exists (a survey on knowledge compilation appears as (12)). Apart from the above cited papers, many cases of compilable and non-compilable problems have been shown in (68; 42); ad hoc complexity classes and reductions for proving compilability and non-compilability have been defined in (13). Nevertheless the syntactic approach considered here is, to the best of our knowledge, original. A syntactic approach has been used in other works, e.g., in (69) for characterizing polynomial and NP-complete sub-cases of existential SO logic over graphs, and in (83) for characterizing approximable NP optimization problems.

This chapter is organized as follows: In Section 7.2 we recall preliminary notions and present a couple of motivating examples, while in Section 7.3 the main theorem concerning queries compilable to P is presented. In Section 7.4 we investigate the reasonableness of the assumptions of such a theorem, showing that, relaxing them, non-compilable problems can be expressed. Section 7.5 generalizes results of Section 7.3 by dealing with compilability to other complexity classes. Several examples will be described throughout the chapter. Discussions and plans future work are given in Section 7.6.

7.2 Preliminaries and motivating examples

In this section we give the preliminary notions that will be used throughout the chapter. First of all, we give the formal definition of *problem compilable to* P; such a definition refers only to problems whose input is split into a fixed and a varying part. Afterwards, we use a couple of specific examples of deduction problems, to highlight the intuitions behind the definitions. In the following, $|\cdot|$ denotes the size of a structure.

Definition 7.2.1 (Problem compilable to P **(14; 114)).** *A problem π, whose input schema is split into a fixed and a varying part, $\vec{F}$ and $\vec{V}$ respectively, is said to be* compilable to P *if there exist two polynomials p_1, p_2 and an algorithm ASK such that for each instance KB of $\vec{F}$ there is a data structure D_{KB} such that:*

1. $|D_{KB}| \leq p_1(|KB|)$;
2. *For each instance δ of $\vec{V}$ the call $ASK(D_{KB}, \delta)$ returns yes if and only if $\langle KB, \delta\rangle$ is a "yes" instance of π;*
3. *$ASK(D_{KB}, \delta)$ requires time $\leq p_2(|\delta| + |D_{KB}|)$.*

The problems we consider in this section refer to propositional logic, and in particular to deduction from a propositional formula in conjunctive normal form (CNF), i.e., a conjunction of clauses, where a clause is a disjunction of literals, i.e., propositional variables, possibly negated.

The problems we consider are *Conjunction inference* and *Clause inference*, whose definitions are the following:

Example 7.2.1 (Conjunction inference (Conj-inf)). *Given a propositional formula T in CNF (fixed part), and a conjunction of literals $J = l_1 \wedge \cdots \wedge l_n$ (varying part), the problem amounts to decide whether $T \models J$, i.e., whether every model of T is also a model of J.*

Example 7.2.2 (Clause inference (Clause-inf)). *Given a propositional formula T in CNF (fixed part), and a disjunction of literals $C = l_1 \vee \cdots \vee l_n$ (varying part), the problem amounts to decide whether $T \models C$, i.e., whether every model of T is also a model of C.*

To make above specifications more clear, consider the following formula T:

$$(\neg c \vee d) \wedge (a \vee \neg b) \wedge (c \vee \neg a) \wedge (\neg d \vee \neg a). \tag{7.1}$$

Formula T logically implies both the conjunction $\neg a$ and the clause $c \vee \neg b$.

As mentioned in Section 7.1, in our setting the problem instance must be represented as a relational database. As for the fixed part, there are several ways to encode a formula in CNF as a database (cf. e.g., Example 1.2.3 and (69)). Differently from Example 1.2.3, here we choose to represent a formula $T = \gamma_1 \wedge \cdots \wedge \gamma_m$ over the set of variables $x_1, \ldots, x_n$ as a directed graph, i.e., a binary relation E_T, in which:

- There is one node for each clause γ_i $(1 \leq i \leq m)$, and one node for each of the literals x_i, $\neg x_i$ $(1 \leq i \leq n)$;

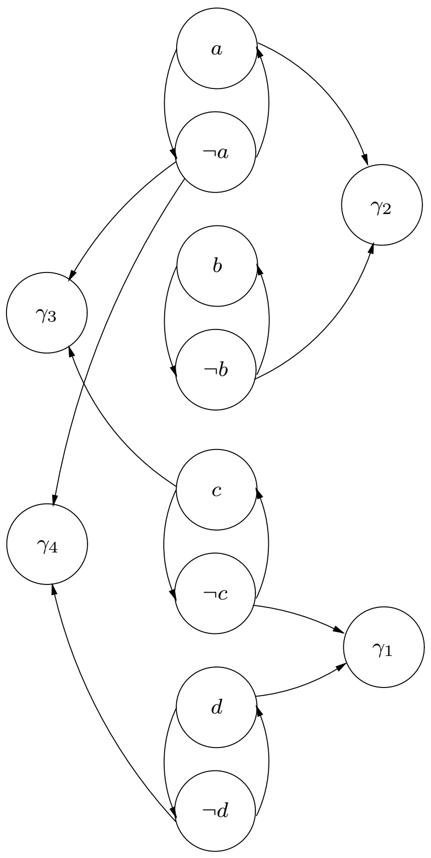

Figure 7.2: Graph of the CNF formula (7.1)

- For each variable x_i there is one edge from x_i to $\neg x_i$ and one edge from $\neg x_i$ to x_i;
- For each clause $\gamma_i = l_i^1 \vee \cdots \vee l_i^{k_i}$, there are k_i edges ending in γ_i, one for each literal $l_i^1, \ldots, l_i^{k_i}$.

Note that clauses correspond to sink nodes in the graph, and that complementary literals form a cycle of length 2. We call a relation like E_T a *fixed* relation. In Figure 7.2 we show the graph corresponding to the CNF formula (7.1).

The varying parts of both problems can be simply represented as monadic relations: C for the clause, and J for the conjunction, storing tuples corresponding to their literals; by referring to the previous example, conjunction $\neg a$ and the clause $c \vee \neg b$ (both of them logically implied by the CNF formula in Figure 7.1), are represented by the following relations:

J

$\neg a$

C

c
$\neg b$

We call relations like J and C *varying* relations.

Now, we have to represent the deduction problems Conj-inf and Clause-inf as queries. Since both problems are coNP-complete, queries must be formulae in the universal fragment of second-order logic (USO), cf. (51). Before showing the complete queries, we go through some intermediate formulae, which are useful for what follows.

A given monadic relation M represents an *interpretation* of a CNF formula T, encoded as described above, represented as relation E_T, if and only if the following first-order formula evaluates to true:

$$\forall X\ \forall Y\ \ (E_T(X,Y) \wedge E_T(Y,X))\ \rightarrow\ (M(X) \leftrightarrow \neg M(Y))\,.$$

The above formula, which will be abbreviated as $int(M, E_T)$, asserts that for each pair of complementary literals X and Y, exactly one is in M (implicitly getting truth value true). By referring again to formula (7.1), a possible interpretation assigns true to a, c, d and false to b, and it is encoded by the following relation M:

M	a
	$\neg b$
	c
	d

Note that this relation makes the previous formula true.

Interpretation M a *model* of T if and only if it assigns true to at least one literal L for each clause C, i.e., for every sink node of E_T. Formally, M is a model of T if and only if the following first-order formula (abbreviated as $model(M, E_T)$) evaluates to true:

$$int(M, E_T)\ \wedge\ \forall X\ \forall Y\ \ (\neg E_T(X,Y)\ \rightarrow\ \exists Z\ \ (E_T(Z,X) \wedge M(Z)))\,. \qquad (7.2)$$

Returning to the example, it can be observed that interpretation M in the above table (that does not satisfy clause c_4 in formula (7.1)) makes formula (7.2) false. On the other hand, the following one (a model of (7.1)) makes it true:

M	$\neg a$
	$\neg b$
	c
	d

Problems Conj-inf and Clause-inf can now be simply represented, respectively, as the following USO queries:

Example 7.2.3 (Conjunction inference (Example 7.2.1 continued)). *Given a propositional formula T in CNF (fixed part), encoded in a binary relation E_T, and a monadic relation J encoding a conjunction of literals $l_1 \wedge \cdots \wedge l_n$ (varying part), an USO specification for the Conjunction inference problem is as follows:*

$$\forall M\ model(M, E_T)\ \rightarrow\ \forall X\ \ (J(X) \rightarrow M(X))\,. \qquad (7.3)$$

Example 7.2.4 (Clause inference (Example 7.2.2 continued)). *Given a propositional formula T in CNF (fixed part), encoded in a binary relation E_T, and a monadic relation C encoding a disjunction of literals $l_1 \vee \cdots \vee l_n$ (varying part), an USO specification for the Clause inference problem is as follows:*

$$\forall M \; model(M,T) \;\rightarrow\; \exists X \;\; (C(X) \wedge M(X))\,. \tag{7.4}$$

It is known (114) that Conj-inf can take advantage of compilation: given a CNF formula T and a conjunction J, off-line reasoning amounts to computing the set of literals logically following from T, i.e., $I = \{l \mid l \text{ is a literal and } T \models l\}$, while on-line reasoning just checks that each literal occurring in J belongs to I. Since the set I has polynomial size in the size of T, and on-line reasoning is polynomial-time in the size of V and J, Conj-inf is said –according to Definition 7.2.1– to be compilable to P.

This idea can be readily used by transforming the USO query (7.3) as follows. Off-line reasoning is captured by computing a *view*, i.e., the following second-order formula open with respect to first-order variable X:

$$infLiteral(X) \;\doteq\; \forall M \; model(M, E_T) \;\rightarrow\; M(X). \tag{7.5}$$

By referring to formula (7.1), since all its models (extensions for relation M that make formula (7.2) true) are:

$$\boxed{\begin{array}{c}\neg a\\ \neg b\\ \neg c\\ \neg d\end{array}} \qquad \boxed{\begin{array}{c}\neg a\\ \neg b\\ \neg c\\ d\end{array}} \qquad \boxed{\begin{array}{c}\neg a\\ \neg b\\ c\\ d\end{array}}$$

it follows that $infLiteral(X)$ would store the following literals:

$$\boxed{\begin{array}{c}\neg a\\ \neg b\end{array}}$$

On-line reasoning is done by evaluating the following query, obtained by *rewriting* (7.3):

$$\forall X \; J(X) \;\rightarrow\; infLiteral(X). \tag{7.6}$$

In our example, the conjunction J is $\neg a$, therefore formula (7.6) is true.

In general, computing the view (7.5) is an NP-hard task, but in KC we don't care about it: what matters is that the view needs polynomial space to be stored, and that the rewritten query (7.6) is first-order, hence polynomial-time.

It is possible to prove that the rewritten query (7.6) is equivalent to the original one (7.3). Actually, in the next section we show a subset of second-order queries – generalizing (7.3) – for which the existence of a polynomial-size view and a first-order equivalent query is guaranteed.

The situation is radically different for the Clause-inf problem, for the following reason. Given a CNF formula T and a clause δ, off-line reasoning could – in principle – be done by computing all clauses that logically follow from T, and on-line reasoning just by checking whether δ is one of them or not. More compactly, only the clauses not implied by other ones, i.e., the so called *prime implicates*, could be computed during off-line reasoning. Anyway, even the latter method is not practical, since it has been proven in (29) that there exist CNF formulae in which the number of prime implicates is exponential in their size.

The definition of non-compilable problem follows from Definition 7.2.1: π is *non-compilable to* P if there are no polynomials p_1, p_2 with the properties stated in Definition 7.2.1. Intuitively, if a polynomial p_1 with the properties mentioned in Definition 7.2.1 does not exist, it means that compilation would take too much (i.e., super-polynomial) space. Dually, if p_1 exists and p_2 does not, then encoding of the problem in a polynomial-space data structure is possible, but it is not possible to extract information from it in polynomial time.

Actually, it is still not known whether the Clause-inf problem is compilable or not, although there are some conditional results that give interesting information. It has been shown (14; 79) that if Clause-inf is compilable, then NP $\subseteq$ P/poly holds, that implies (77) that $\Sigma_2^p = \Pi_2^p$, i.e., the Polynomial Hierarchy collapses, an event that – although not as strong as P = NP – is widely conjectured to be false. A similar result can be proven for the more general problem $T \models W$, where W is any propositional formula. If this problem is compilable, then P = NP, as tautologies could be detected in polynomial time compiling the empty T.

Nevertheless, to give a formal proof that a problem is not compilable is quite difficult. As an example, proving that Clause-inf is not compilable implies P $\neq$ NP (a necessary and sufficient condition for compilability is shown in (14)). In fact, this is a typical situation when dealing with non-compilability of problems, that we can parallel with the traditional notion of polynomial-time intractability: proving intractability is usually done proving NP-hardness, which means that a problem does not have polynomial-time algorithms provided P $\neq$ NP.

Summing up, proving compilability can be done by exhibiting polynomials according to Definition 7.2.1, while non-compilability is proven only conditionally. Hence, in the following we will use the term “non-compilable problem”, by meaning this conditional result (i.e., compilability would imply the collapse of the PH).

7.3 Queries compilable to P

In this section we present the main result of the chapter concerning compilability. We give a sufficient condition for compilability to P in terms of syntactic restrictions of second-order formulae.

In the following, $\delta(\vec{X})$ denotes that all free first-order variables in formula δ are in the list $\vec{X}$. Moreover, $\delta(\vec{M})$ denotes that predicate symbols in the list $\vec{M}$ may occur in δ.

First of all, we present a result that will be useful in the following.

Lemma 7.3.1. *Let ψ be a second-order query of the form:*

$$\forall \vec{M}\ \alpha(\vec{M})\ \vee\ \forall \vec{X}\ \left(\beta(\vec{X})\ \vee\ \gamma(\vec{X}, \vec{M})\right), \tag{7.7}$$

where α is an arbitrary second-order formula in which no free first-order variables *occur, and β and γ are arbitrary first-order formulae, with β not containing predicates in $\vec{M}$.*

Formula (7.7) *is equivalent to:*

$$\forall \vec{X}\ \beta(\vec{X})\ \vee\ view(\vec{X}), \tag{7.8}$$

where $view(\vec{X})$ is defined as:

$$\forall \vec{M}\ \alpha(\vec{M})\ \vee\ \gamma(\vec{X}, \vec{M}). \tag{7.9}$$

The proof is delayed to Appendix F. Lemma 7.3.1 captures several intuitions:

- Complex (possibly quantified) subformulae can be treated as single components, like α, β, and γ.
- This is very useful for modelling the off-line computation, that may involve all predicates, except for the on-line ones. As an example, in formula (7.3) concerning Conj-inf, the off-line computation involves $model(M, E_T)$ and $M(X)$, cf. (7.5).
- In query (7.7) it is possible to separate the second-order and fixed parts from the varying part. This allows us to process the former in the off-line phase (by means of the view (7.9)), and to evaluate the rewritten query (7.8) in the on-line phase.

In what follows, we make the assumption that the size of the query is fixed with respect to the size of the database, which is implied by the *data complexity* (cf. (128)) assumption. As a consequence, the arity of $\vec{X}$ is a constant, and polynomial space in the size of data is enough to represent $view(\vec{X})$. Furthermore, since (7.8) is an ordinary first-order formula, its evaluation requires polynomial time in the size of data.

From these observations, the main theorem follows:

Theorem 7.3.1. *Let ψ, α, β, γ, and view as in Lemma 7.3.1. If neither α nor γ contain varying predicates, then the problem specified by ψ is compilable to* P. *In particular, off-line reasoning is done by computing the view* (7.9)*, and on-line reasoning by evaluating the (rewritten) query* (7.8).

The above theorem is stated for second-order queries starting with universal quantification, which are adequate for modeling deduction problems: in such problems, typically, a formula must be true in *all* models of a specified set. A dual theorem for queries starting with existential quantification (adequate, for example, for model checking problems) holds.

7.3.1 Examples

In this subsection we show some applications of the above theorem.

From coNP to P: Deduction problems

Theorem 7.3.1 immediately applies to the Conj-inf problem (cf. Section 7.2), by assuming $\vec{M} = \{M\}$, $\vec{X} = \{X\}$, and:

- $\alpha(\vec{M}) = \neg model(M, E_T)$;
- $\beta(\vec{X}) = \neg J(X)$;
- $\gamma(\vec{X}, \vec{M}) = M(X)$.

A generalization of Conj-inf with the same fixed part is *2CNF inference*, in which the varying part is a formula in CNF with exactly two literals per clause (2CNF).

Example 7.3.1 (2CNF inference (2CNF-inf)). *Given a propositional formula T in CNF (fixed part), encoded in a binary relation E_T, and a propositional formula F in CNF exactly two literals per clause (2CNF), the problem amounts to decide whether $T \models F$, i.e., whether every model of T also a model of F.*

Representing a 2CNF formula F can be done simply by means of a binary (symmetric) relation R_F whose tuples correspond to the clauses in F. As an example, by referring to formula (7.1), the logically implied 2CNF formula $(\neg a \vee \neg c) \wedge (\neg b \vee c)$ can be encoded by the following relation:

R_F

$\neg a$	$\neg c$
$\neg c$	$\neg a$
$\neg b$	c
c	$\neg b$

The problem $T \models F$ corresponds to the following USO formula:

$$\forall M \; model(M, E_T) \;\rightarrow\; \forall X_1, X_2 \;\; (R_F(X_1, X_2) \;\rightarrow (M(X_1) \;\vee\; M(X_2))) \qquad (7.10)$$

that evaluates to true if and only if every model M of T contains at least one literal of every clause of F.

In this case the view is different: In particular, $\vec{M} = \{M\}$, $\vec{X} = \{X_1, X_2\}$, and Theorem 7.3.1 applies with:

- $\alpha(\vec{M}) = \neg model(M, E_T)$;
- $\beta(\vec{X}) = \neg R_F(X_1, X_2)$;
- $\gamma(\vec{X}, \vec{M}) = M(X_1) \vee M(X_2)$.

As noted in (99), the generalization *kCNF inference* of 2CNF-inf, in which the varying part is a CNF formula with exactly k literals per clause, is still compilable, as long as k is fixed with respect to the size of T. Query (7.10) can be modified to account for longer clauses simply by increasing the arity of R_F, and Theorem 7.3.1 still applies.

From coNP **to** P**: Graph coloring**

Theorem 7.3.1 can be applied to non-deductive problems. As an example, we consider the following *Forced colors in a graph* problem:

Example 7.3.2 (Forced colors in a graph). *Given a graph (fixed part), encoded in a binary relation edge (cf. Example 1.2.2), and a set of nodes (varying part) encoded in a monadic relation S, the problem amounts to decide whether every node in S is forced to be assigned a pre-determined color (e.g., red) in each 3-coloring of the graph in which two fixed nodes, a and b, have a given color, e.g., red and green, respectively.*

Using unary predicates R, G, and B for the colors, in Example 1.2.2 we already showed a closed first-order formula, denoted here by $3col(R, G, B, edge)$, that represents the fact that R, G, and B encode a 3-coloring of graph represented by *edge*. The query for the Forced colors problem is as follows:

$$\forall R, G, B \; (3col(R, G, B, edge) \wedge R(a) \wedge G(b)) \; \rightarrow \; \forall X \;\; (S(X) \rightarrow R(X)) \,. \qquad (7.11)$$

Theorem 7.3.1 applies when $\vec{M} = \{R, G, B\}$, $\vec{X} = \{X\}$, and:

- $\alpha(\vec{M}) = \neg(3col(R, G, B, edge) \wedge R(a) \wedge G(b))$;
- $\beta(\vec{X}) = \neg S(X)$;
- $\gamma(\vec{X}, \vec{M}) = R(X)$.

The intuitive meaning of the forced colors problem is to check whether choosing the color for a couple of nodes constrains a different set S of nodes (not known in advance) to have a specific color. Such a problem is an abstraction for several resource allocation problems, e.g., the *Frequency assignment* problem in the context of telecommunications networks. In the Frequency assignment problem nodes are transmitters, colors are frequencies, and an edge means that two nodes cannot be assigned the same frequency. In the query (7.11) we are basically asking whether assigning frequencies to a subset of the transmitters implies that other nodes (set S) must have a specific frequency. Clearly, query (7.11) can be generalized to any constant number of colors (frequencies).

As another example in the context of graphs we consider the *Few blue neighbors* problem:

Example 7.3.3 (Few blue neighbors). *Given a graph (fixed part), encoded in a binary relation edge (cf. Example 1.2.2), and a set of nodes (varying part), encoded in a monadic relation T, the problem amounts to decide whether every node in T has at most one blue neighbor in each 3-coloring of the graph in which two fixed nodes, a and b, have a given color, e.g., red and green, respectively.*

A query for the few blue neighbors problem is the following:

$$\begin{array}{l}\forall R, G, B \; (3col(R, G, B, edge) \wedge R(a) \wedge G(b)) \rightarrow \\ \qquad\qquad \forall X, Y, Z \; (T(X) \wedge edge(X, Y) \wedge edge(X, Z) \wedge B(Y) \wedge B(Z)) \; \rightarrow \; Y = Z.\end{array}$$

In this case, $\vec{M} = \{R, G, B\}$, $\vec{X}$ is $\{X, Y, Z\}$, and:

- $\alpha(\vec{M}) = \neg 3col(R, G, B, edge)$;
- $\beta(\vec{X}) = \neg T(X)$;
- $\gamma(\vec{X}, \vec{M}) = \neg edge(X, Y) \vee \neg edge(X, Z) \vee \neg B(Y) \vee \neg B(Z) \vee (Y = Z)$.

Also the few blue neighbors problem abstracts resource allocation problems. As an example it can be viewed as a variant of the frequency assignment problem, where we want transmitters in the set T to have at most one neighbor with a given frequency, because it may interfere with others.

From Π_2^p to P: Circumscription

In principle, we can apply Theorem 7.3.1 to any problem in the PH. As an example, we consider the *Conjunction inference from circumscription* problem, defined as follows:

Example 7.3.4 (Conjunction inference from circumscription (Conj-circ)). *Given a propositional formula T in CNF (fixed part) encoded in binary relation E_T, and conjunction of literals $J = l_1 \wedge \cdots \wedge l_n$, the problem amounts to decide whether $CIRC(T) \models J$, i.e., whether every model of the* circumscription *of T is also a model of J.*

Circumscription (88) is a form of non-monotonic reasoning, and the models of $CIRC(T)$ are the *minimal* models of T, with respect to set containment (as an example, the minimal models of $a \vee b$ are $\{a\}$ and $\{b\}$, and the model $\{a, b\}$ is not minimal). As shown in (50), Conj-circ (ignoring the distinction between varying and fixed parts) is Π_2^p-complete. We now show that it can be compiled to P. First of all we introduce the following USO formula (abbreviated as $minModels(M, E_T)$) that evaluates to true if and only if M represents a minimal model of T:

$$\begin{array}{lll} model(M, E_T) \wedge & & \\ \forall N\ model(N, E_T) & \rightarrow & \neg[(\forall X\ var_T(X) \wedge N(X) \rightarrow M(X)) \wedge \\ & & (\exists X\ var_T(X) \wedge M(X) \wedge \neg N(X))] \end{array} \tag{7.12}$$

where var_T is a unary fixed relation storing the propositional variables occurring in T. Now, Conj-circ can be specified as follows:

$$\forall M\ minModels(M, E_T) \ \rightarrow\ (\forall X\ J(X) \rightarrow M(X)),$$

and Theorem 7.3.1 applies with $\vec{M} = \{M\}$, $\vec{X} = \{X\}$, and:

- $\alpha(\vec{M}) = \neg minModels(M, E_T)$;
- $\beta(\vec{X}) = \neg J(X)$;
- $\gamma(\vec{X}, \vec{M}) = M(X)$.

It is interesting to notice that off-line reasoning, i.e., to compute the view $\forall M\ minModels(M, E_T) \rightarrow M(X)$ is a Π_2^p-hard task. In this case, KC is even more convenient, since a Π_2^p-complete problem is transformed into a polynomial one.

7.4 Queries non-compilable to P

In this section we deal with non-compilability, in the conditional sense of Section 7.2. More precisely, we show the reasonableness of the assumptions of Theorem 7.3.1 concerning γ and α, by showing that the set of second-order queries in which either assumption does not hold contains queries non-compilable to P.

Theorem 7.4.1. *The set of formulae of Lemma 7.3.1 in which in γ there are varying predicates, contains queries non-compilable to* P.

Proof. By allowing formulae with varying predicates in γ, Clause-inf (cf. Example 7.2.4) can be modelled by taking $\vec{M} = \{M\}$, $\vec{X} = \{\}$, and:

- $\alpha(\vec{M}) = \neg model(M, E_T)$;
- $\beta(\vec{X}) = \mathsf{false}$;
- $\gamma(\vec{X}, \vec{M}) = \exists Y\ C(Y) \wedge M(Y)$.

We remind that this problem, as mentioned in Section 7.2, is non-compilable to P. □

Before claiming the next result, we briefly describe a generalized version of circumscription (cf. Section 7.3.1), and the *Conjunction inference from* $(P;Z)$-*circumscription* problem, useful for what follows.

Example 7.4.1 (Conjunction inference from (P;Z)-circumscription (Conj-(P;Z)-circ)). *Given a propositional formula T in CNF (fixed part), encoded in a binary relation E_T, and conjunction of literals $l_1 \wedge \cdots \wedge l_n$ (varying part), encoded in a monadic relation J, the problem amounts to decide whether $CIRC(T;P;Z) \models J$, i.e., whether every $(P;Z)$-minimal model of T also a model of J.*

In the $(P;Z)$-*circumscription* only a subset of the propositional variables (those in a specified set P) are minimized, while the remaining ones (in the complementary set Z) are left *floating* (cf. (88)). This leads to the notion of $(P;Z)$-*minimal* models, which coincide with minimal models described in Section 7.3.1 when $Z = \emptyset$. As an example, if $P = \{b\}$ and $Z = \{a\}$, then $a \vee b$ has only one $(P;Z)$-minimal model, i.e., $\{a\}$ (the minimal model $\{b\}$ is not $(P;Z)$-minimal).

The USO formula (abbreviated as $minModels_{(P;Z)}(M, P, E_T)$) that evaluates to true if and only if M represents a $(P;Z)$-minimal model of T can be obtained by modifying formula (7.12) as follows:

$$\begin{aligned} & model(M, E_T) \wedge \\ & \forall N\ model(N, E_T) \ \rightarrow\ \neg[(\forall X\ P(X) \wedge N(X) \rightarrow M(X)) \wedge \\ & \qquad\qquad\qquad\qquad\qquad (\exists X\ P(X) \wedge M(X) \wedge \neg N(X))] \end{aligned} \tag{7.13}$$

where, P is a unary relation storing all propositional variables to be minimized.

Hence, Conj-$(P;Z)$-circ can be immediately specified with the query:

$$\forall M\ minModels_{(P;Z)}(M, P, E_T)\ \rightarrow (\forall X\ J(X) \rightarrow M(X)). \tag{7.14}$$

We are now ready to present the next theorem.

Theorem 7.4.2. *The set of formulae of Lemma 7.3.1 in which in α there are varying predicates, contains queries non-compilable to* P.

Proof. Consider the Conj-(P;Z)-circ problem with the P set varying; by allowing formulae of kind (7.7) with varying predicates in α, the query (7.14) matches conditions of Lemma 7.3.1 by taking $\vec{M} = \{M\}$, $\vec{X} = \{X\}$, and:

- $\alpha(\vec{M}) = \neg minModels_{(P;Z)}(M, P, E_T)$;
- $\beta(\vec{X}) = \neg J(X)$;
- $\gamma(\vec{X}, \vec{M}) = M(X)$.

However, it has been proven in (14) that in such a case $CIRC(T; P; Z) \models J$ (J being a conjunction of literals) is non-compilable to P, even when T is a positive 2CNF formula and J is a single literal. $\square$

We remark that the above theorems just deal with syntactic criteria. Their goal is not to provide a sharp separation between compilable and non-compilable problems, but rather to justify the assumptions of Theorem 7.3.1.

The similarity between Conj-inf (7.3) and Clause-inf (7.4) deserves further comments. In general, a second-order formula can be put in prenex form, and its matrix in CNF, treating arbitrary second-order subformulae where no free first-order variables occur (like α in Lemma 7.3.1) as single literals. A formula in this form is said to be in *prenex generalized CNF* (PGCNF). In all compilability examples of Section 7.3, queries are actually in PGCNF, and their matrices contain exactly one clause. On the other hand, we note that Clause-inf (7.4) is also in PGCNF, but has two clauses.

It is actually possible to show PGCNF queries with two clauses, which are compilable to P. As an example, the next query combines the two compilable problems of Section 7.3.1, having the same fixed part:

$$\begin{aligned}&\forall R, G, B\ (3col(R, G, B, E_H) \wedge R(a) \wedge G(b)) \ \rightarrow \\ &\qquad [\forall X, Y, Z\ (T(X) \wedge E_H(X, Y) \wedge E_H(X, Z) \wedge B(Y) \wedge B(Z) \rightarrow Y = Z) \ \wedge \\ &\qquad \forall W\ (S(W) \rightarrow R(W))].\end{aligned}$$

The above query is of the form:

$$\forall \vec{M} \left\{ \alpha(\vec{M}) \ \vee \ \left[(\forall \vec{X}_1\ \beta(\vec{X}_1) \ \vee \ \gamma(\vec{X}_1, \vec{M})) \ \wedge \ (\forall \vec{X}_2\ \beta(\vec{X}_2) \ \vee \ \gamma(\vec{X}_2, \vec{M})) \right] \right\},$$

and all problems of this kind are indeed compilable to P, as shown by the next theorem, that generalizes Theorem 7.3.1.

Theorem 7.4.3 ($\wedge$-composition). *Let ψ be a second-order query of the form:*

$$\forall \vec{M} \ \bigwedge_i \left\{ \alpha_i(\vec{M}) \ \vee \ \forall \vec{X}_i\ (\beta_i(\vec{X}_i) \ \vee \ \gamma_i(\vec{X}_i, \vec{M})) \right\}, \tag{7.15}$$

where α_i are arbitrary second-order formulae where no free first-order variables occur, β_i and γ_i are arbitrary first-order formulae, with β_i not containing predicates in $\vec{M}$, and α_i, γ_i do not contain varying predicates.

The problem specified by ψ is compilable to P. *In particular, off-line reasoning is done by computing the views $view_i(\vec{X}_i)$, defined as:*

$$\forall\vec{M}\ \alpha_i(\vec{M})\ \vee\ \gamma_i(\vec{X}_i, \vec{M}),$$

and on-line reasoning can be done by evaluating the (rewritten) query:

$$\bigwedge_i \left(\forall\vec{X}_i\ \beta_i(\vec{X}_i)\ \vee\ view_i(\vec{X}_i)\right).$$

Proof. Because of the presence of the universal quantifier and the outermost connective $\wedge$, by pushing $\forall\vec{M}$ in, formula (7.15) can be safely rewritten as $\bigwedge_i \psi_i$ where each ψ_i is of the form (7.7). Now, query ψ can be rewritten using the same arguments of the proof of Lemma 7.3.1. $\square$

The above theorem says that, when we have two or more problems that are "independently" compilable to P, then their combination is compilable to P as well. Obviously, the theorem does not apply to the Clause-inf problem, because in (7.4) the scope of the first-order variable X spans both clauses of the matrix. In particular, it seems that the presence of the same first-order variable in both a fixed and a varying predicate in different clauses is a plausible predictor for non-compilability of a query.

7.5 Queries compilable to other classes

In general, KC deals with making on-line reasoning more efficient, but not necessarily polynomial-time. Hence, a definition more general than Definition 7.2.1 is that of *problem compilable to* C, where C is an arbitrary complexity class:

Definition 7.5.1 (Problem compilable to C**).** *A problem π, whose input schema is split into a fixed and a varying part, $\vec{F}$ and $\vec{V}$ respectively, is said to be* compilable to C *if there exists a polynomial p and an algorithm ASK such that for each instance KB of $\vec{F}$ there is a data structure D_{KB} such that:*

1. *$|D_{KB}| \leq p(|KB|)$;*
2. *for each instance δ of $\vec{V}$ the call $ASK(D_{KB}, \delta)$ returns yes if and only if $\langle KB, \delta\rangle$ is a "yes" instance of π;*
3. *$ASK(D_{KB}, \delta)$ is in the complexity class* C.

As an example, it makes sense to compile a problem that is Π_2^p-complete to a coNP-complete one, if it is non-compilable to P. To this end, Theorem 7.3.1 is useless, because it only deals with compilation to P. On the other hand, if we can modularize a second-order query as a formula of the form:

$$\vec{Q}_{SO}\vec{M}\ \vec{Q}_{FO}\vec{X}\ \eta(\vec{M}, \vec{X}, \lambda(\vec{X})) \tag{7.16}$$

where $\vec{Q}_{SO}$ and $\vec{Q}_{FO}$ are, respectively, lists of second-order and first-order quantifiers, λ is an arbitrary second-order formula in which neither on-line predicates nor predicates in $\vec{M}$ occur, and η is second-order, then the evaluation of $\lambda(\vec{X})$ can be done off-line.

Several interesting deduction problems can indeed be specified in such a form. As an example, another variant of the Clause-inf problem is the Clause-gcwa problem, defined as follows:

Example 7.5.1 (Clause inference in the Generalized closed-world assumption (Clause-gcwa)). *Given a propositional formula T in CNF (fixed part), encoded as a binary relation E_T, and disjunction of literals $l_1 \vee \ldots \vee l_n$ (varying part), encoded as a monadic relation C, the problem amounts to decide whether $GCWA(T) \models C$, i.e., whether every model of $GCWA(T)$ is also a model of C.*

The *generalized closed-world assumption* (GCWA) of a propositional formula T is a form of non-monotonic reasoning defined in (98) as follows:

$$GCWA(T) = T \cup \{\neg x \mid \text{variable } x \text{ is } \mathsf{false} \text{ in all minimal models of } T\}.$$

As an example, the GCWA of formula $T = a \vee b$ is T itself, since both a and b are true in at least one minimal model of T.

As shown in (50), Clause-gcwa is Π_2^p-complete; moreover, if T is fixed, compilation to P is not possible, cf. (14). There is some room for compilation to an intermediate class, such as coNP, using the following idea. All negative literals that are true in all minimal models of T can be computed during the off-line phase, and can obviously be represented in a polynomial-size structure. On-line reasoning will use such literals as well as the original formula T, and can be done by means of a coNP computation.

The following view $gcwaLit(Y)$ represents off-line reasoning, because it stores all negative literals Y that are true in all minimal models of T:

$$\forall X \ \ (var_T(X) \wedge E_T(X,Y) \wedge E_T(Y,X)) \ \rightarrow \ (\forall N \ minModels(N, E_T) \rightarrow N(Y))$$

where var_T is a unary fixed relation storing the propositional variables occurring in T. Note that Y is a negative literal, because it is the negation of X, that is a variable of T. Now, Clause-gcwa can be written as:

$$\forall M \ \ (model(M, E_T) \ \wedge \ (\forall Y \ gcwaLit(Y) \rightarrow M(Y))) \ \rightarrow \ (\exists X \ C(X) \wedge M(X))\,,$$

that is indeed in the form (7.16), with $\lambda(\vec{X}) = gcwaLit(Y)$.

Another interesting problem specifiable by means of a formula of the form (7.16) is the WIDTIO approach to knowledge base revision (132). In this case, the problem is to check whether a clause q follows from a formula T revised with another formula p that makes T inconsistent. The view is the set of clauses in T which "survive" after the revision. The Π_2^p-complete query can be compiled to a coNP-complete one.

It is worth noting that formulae of the kind (7.16) subsume those of Theorem 7.3.1. They are also less precise, because they capture formulae like (7.4): just take $\lambda(\vec{X})$ as the empty formula.

7.6 Discussion and future research directions

In this chapter we have focused on Knowledge Compilation in the context of relational databases. Both fixed and varying parts of the instance are represented, as usual, as relations, and the inference operator is represented as a query in second-order logic. Our main result concerning compilability is the definition of a syntactic restriction of second-order formulae that guarantees the existence of a polynomial-size view and a first-order equivalent rewritten query, thus allowing on-line reasoning to be done in polynomial time.

We have applied such a result to a variety of deduction problems, with underlying complexity at various levels of the Polynomial Hierarchy. Non-deductive problems concerning graphs have also been addressed. The syntactic restriction has been motivated by showing that relaxing it we obtain a set of second-order queries containing queries non-compilable to P.

Finally, we have addressed the issue of compilability to higher classes in the Polynomial Hierarchy, by showing examples of Π_2^p-complete deduction problems which are compilable to coNP-complete ones.

Our approach is somewhat limited because our target language (for problems compilable to P) is first-order logic. This implies that some compilable problems, like, e.g., *Cycles in a Hamiltonian reduction*, defined in (12), cannot be captured, because they involve on-line reasoning which is not first-order expressible. To this end, we plan to investigate extensions of the target language to classes of formulae in fixpoint logic.

We also aim to characterize in a more detailed way the "frontier" between compilable and non-compilable problems, e.g., by adding finer-grained constraints to the conditions of Theorems 7.4.1 and 7.4.2.

Results in this chapter have been published in (18).

Chapter 8

The *NP-Alg* and ConSql languages

8.1 Introduction

The efficient solution of NP-hard combinatorial problems, such as resource allocation, scheduling, planning, etc., is becoming crucial also for applications of industrial relevance. Currently, this task is accomplished either by using ad-hoc hand-written programs, or by calling, from general-purpose programming languages, specialized libraries for expressing constraints, e.g., (73). A promising alternative is going to be made by declarative constraint modelling languages (cf. Section 1.1), e.g., OPL.

However, in all cases, data encoding the instance are either in text files in an ad-hoc format, or in standard relational DBs accessed through libraries called from external programs (cf., e.g., (74)). In other words, there is not a strong integration between the data definition and the constraint modelling and programming languages.

Indeed, such an integration is particularly needed in industrial environments, where the necessity for solving combinatorial problems coexists with the presence of large databases where data to be processed lie. Hence, constraint solvers that operate externally to the databases of the enterprise may lead to a series of disadvantages, first of all a potential lack of the integrity of recorded data. To this end, a better coupling between standard data repositories and constraint solving engines is highly desiderable.

In this chapter we tackle exactly this issue, discussing the integration of constraint modelling and programming into relational database management systems (R-DBMSs). In particular, we show how standard query languages for relational databases can be extended in order to give them constraint modelling and solving capabilities: with such languages, constraint problem specifications can be viewed just like (more complex) queries to standard data repositories. In what follows, we propose extensions of standard query languages such as relational algebra and SQL, that are able to formulate queries defining combinatorial and constraint problems.

In principle relational algebra can be used as a language for testing constraints. As an example, given relations A and B, testing whether all tuples in A are contained in B can be done by computing the relation $A - B$, and then checking its emptiness.

Anyway, it must be noted that relational algebra is unfeasible as a language for expressing NP-hard problems, since it is capable of expressing just a strict subset of the polynomial-time queries (cf., e.g., (2)). As a consequence, an extension is needed.

The proposed generalization of relational algebra is named *NP-Alg*, and it is proven to be capable of expressing all problems in the complexity class NP. We focus on NP because this class contains the decisional versions of most combinatorial problems of industrial relevance (62).

NP-Alg is relational algebra plus a simple *guessing* operator, that declares a set of relations to have an arbitrary extension. Algebraic expressions are used to express constraints. Several interesting properties of *NP-Alg* are provided: its data complexity is shown to be NP-complete, and for each problem π in NP we prove that there is a fixed query that, when evaluated on a database representing an instance of π, solves it. Combined complexity is also addressed.

Since *NP-Alg* expresses all problems in NP, an interesting question is whether a query corresponds to an NP-complete or to a polynomial-time problem. We give a partial answer to it, by exhibiting some syntactical restrictions of *NP-Alg* with polynomial-time data complexity.

In the same way, ConSql (sql with constraints) is the proposed non-deterministic extension of sql, the well-known language for querying relational databases (123), having the same expressive power of *NP-Alg*, and supporting also the specification of optimization problems. We believe that writing a ConSql query for the solution of a combinatorial optimization problem is only moderately more difficult than writing sql queries for a standard database application. The advantage of using ConSql is twofold: it is not necessary to learn a completely new language or methodology, and integration of the problem solver with the information system of the enterprise can be done very smoothly. The effectiveness of both *NP-Alg* and ConSql as constraint modelling languages is demonstrated by showing several queries that specify combinatorial and optimization problems.

The structure of this chapter is as follows. Syntax and semantics of *NP-Alg* are introduced in Section 8.2. Some examples of *NP-Alg* queries for the specification of NP-complete combinatorial problems are proposed in Section 8.3. Main computational properties of *NP-Alg*, including data and combined complexity, expressive power, and polynomial fragments, are presented in Section 8.4. Section 8.5 contains some details of ConSql and its implementation ConSql simulator, as well as the specification of some real-world combinatorial and optimization problems. Finally, Section 8.6 contains conclusions as well as references to main related work.

8.2 *NP-Alg*: Syntax and semantics

We refer to a standard definition of relational algebra with the five operators $\{\sigma, \pi, \times, -, \cup\}$ (2). Other operators such as "$\bowtie$" and "/" can be defined as usual. Attributes (fields) of relations will be denoted either by their names or by their indexes. As an example, given a relation $R(a, b)$, the selection of tuples in R with the same values for the two attributes will be denoted in one of the following forms: $\underset{R.a=R.b}{\sigma}(R)$, $\underset{a=b}{\sigma}(R)$ (since there is no confusion to what relation a and b refer to), or $\underset{\$1=\$2}{\sigma}(R)$. As for join conditions, they will have atoms of the form $a = b$ or $a \neq b$ where a is an attribute name (or even index) of the relation on the left of the join symbol, and b one of that on the right. Finally, temporary relations such as $T = algexpr(\ldots)$ will be used to make expressions easier to read. As usual (cf., e.g., (28)) queries are defined as mappings which are partial recursive and generic, i.e., constants are uninterpreted.

Let D denote a finite relational database, $\vec{R}$ its relational schema, and DOM the unary relation representing the set of all constants occurring in D.

Definition 8.2.1 (Syntax of *NP-Alg*). *An NP-Alg expression has two parts:*

1. *A set* $\vec{Q} = \{Q_1^{(a_1)}, \ldots, Q_n^{(a_n)}\}$ *of new relations of arbitrary arity, denoted as Guess* $Q_1^{(a_1)}, \ldots, Q_n^{(a_n)}$. *Sets* $\vec{R}$ *and* $\vec{Q}$ *must be disjoint.*

2. *An ordinary expression exp of relational algebra on the new database schema* $[\vec{Q}, \vec{R}]$.

For simplicity, until Section 8.4 we focus on *boolean queries*, i.e., queries that admit a yes/no answer. For this reason we restrict *exp* to be a relation that we call *FAIL*.

Definition 8.2.2 (Semantics of *NP-Alg*). *The semantics of an NP-Alg expression is as follows:*

1. *For each possible extension ext of the relations in* $\vec{Q}$ *with elements in DOM, the relation FAIL is evaluated, using ordinary rules of relational algebra.*

2. *If there is an extension ext such that the expression for FAIL evaluates to the empty relation "$\emptyset$" (denoted as FAIL$\Diamond\emptyset$), the* answer *to the boolean query is "yes". Otherwise the* answer *is "no".*

   When the answer is "yes", the extension of relations in $\vec{Q}$ *is a* solution *for the problem instance.*

A trivial implementation of the above semantics obviously requires exponential time, since there are exponentially many possible extensions of the relations in $\vec{Q}$. Anyway, as we will show in Section 8.4.3, some polynomial-time cases indeed exist.

The reason why we focus on a relation named *FAIL* is that, typically, it is easy to specify a decision problem as a set of constraints (cf. forthcoming Sections 8.3 and 8.5). As a consequence, an instance of the problem has a solution if and only if there is an arbitrary choice of the guessed relations such that all constraints are satisfied, i.e., $FAIL = \emptyset$. A $FOUND^{(1)}$ query can be anyway defined as $FOUND = DOM - \underset{\$1}{\pi}(DOM \times FAIL)$. In this case, the answer is "yes" if and only if there is an extension *ext* such that $FOUND \neq \emptyset$.

8.3 Examples of *NP-Alg* queries

In this section we show the specifications of some NP-complete problems, as queries in *NP-Alg*. All examples are on uninterpreted structures, i.e., on unlabeled directed graphs, because we adopt a pure relational algebra with uninterpreted constants. As a side-effect, the examples show that, even in this limited setting, we are able to emulate bounded integers and ordering. This is very important, because the specification of very simple combinatorial problems requires bounded integers and ordering.

In Section 8.5 we use the full power of ConSql to specify some real-world problems.

In what follows, we assume a directed graph is represented as a pair of relations $NODES^{(1)}(n)$ and $EDGES^{(2)}(from, to)$ (with tuples in $EDGES^{(2)}$ having components in $NODES^{(1)}$, hence, $DOM = NODES$).

Example 8.3.1 (Graph k-coloring). *This problem has been already presented as Example 1.2.1. Given a graph as input, we say that it is k-*colorable *if there is a* k*-partition $Q_1^{(1)}, \dots, Q_k^{(1)}$ of its nodes, i.e., a set of k sets such that:*

- $\forall i \in [1,k], \forall j \in [1,k], j \neq i \rightarrow Q_i \cap Q_j = \emptyset$,
- $\bigcup_{i=1}^{k} Q_i = NODES$,

and each set Q_i has no pair of nodes linked by an edge. The problem can be specified in NP-Alg as follows:

$$Guess\ Q_1^{(1)}, \dots, Q_k^{(1)}; \tag{8.1}$$

$$FAIL\_DISJOINT = \bigcup_{i \neq j \in \{1,\dots,k\}} Q_i \bowtie Q_j; \tag{8.2}$$

$$FAIL\_COVER = NODES\ \Delta\ \bigcup_{i=1}^{k} Q_i; \tag{8.3}$$

$$FAIL\_PARTITION = FAIL\_DISJOINT\ \cup\ FAIL\_COVER; \tag{8.4}$$

$$FAIL\_COLORING = \pi_{\$1} \left[\bigcup_{i=1}^{k} \left(\left(\sigma_{\$1 \neq \$2} (Q_i \times Q_i) \right) \underset{\substack{\$1 = EDGES.from \\ \$2 = EDGES.to}}{\bowtie} EDGES \right) \right]; \tag{8.5}$$

$$FAIL = FAIL\_PARTITION\ \cup\ FAIL\_COLORING. \tag{8.6}$$

Expression (8.1) *declares* k *new relations of arity 1. Expression* (8.6) *collects all constraints a candidate coloring must obey to:*

- (8.2) *and* (8.3) *make sure that $Q_1, \dots, Q_k$ is a partition of NODES ("Δ" is the symmetric difference operator, i.e., $A\ \Delta\ B = (A - B) \cup (B - A)$, useful for testing equality since $A\ \Delta\ B = \emptyset \iff A = B$).*
- (8.5) *checks that each set Q_i has no pair of nodes linked by an edge.*

As an example, let $k = 3$ and the database be as follows:

NODES

n
a
b
c
d
e

EDGES

from	*to*
a	*b*
a	*c*
a	*d*
b	*d*
b	*e*

The above extension for relations NODES and EDGES encodes the graph in Figure 2.1. An extension of Q_1, Q_2 and Q_3 such that $FAIL = \emptyset$ is:

Q_1

d

Q_2

a
e

Q_3

b
c

Note that such an extension constitutes a solution of the coloring problem.

We observe that in the specification above the *FAIL_PARTITION* relation (8.4) makes sure that an extension of $Q_1^{(1)}, \ldots, Q_k^{(1)}$ is a *k-partition* of *NODES*. Such an expression can be very useful for the specification of problems, so we introduce, as we did for ESO (cf. Appendix A), an expression:

$$\mathit{failPartition}^{(1)}(N^{(k)}, P_1^{(k)}, \ldots, P_n^{(k)}),$$

that returns an empty relation if and only if $\{P_1^{(k)}, \ldots, P_n^{(k)}\}$ is a partition of $N^{(k)}$. The prefix *fail* in the name of the expression reminds that it should be used in checking constraints. We note that the arity of *failPartition* can, without loss of generality, be fixed to 1, since we can always project out the remaining columns. Other useful syntactic sugar will be introduced in the following examples, and is summarized in Appendix G.1.

Example 8.3.2 (Independent set (62, Prob. GT20, p. 194)). *Given a graph as input, and a positive integer $k \leq |NODES|$, encoded by a relation $K^{(1)}$ containing exactly k tuples, this* NP*-complete problem amounts to decide whether there exists a subset N of NODES, with $|N| \geq k$ that contains no pair of nodes linked by an edge.*

In order to be able to compare the size of N with the integer k (i.e., with the size of relation K), we will use the same trick used in ESO (cf. Appendix A), i.e., we check the existence of an appropriate function *from N to K. To this end, we can define relational algebra expressions that check whether a relation $FUN^{(d+r)}$ is a (total, injective, surjective, or bijective)* function *from domain $D^{(d)}$ to range $R^{(r)}$, as Appendix G.1 shows.*

Hence, the following NP-Alg query specifies the Independent set problem:

$$\begin{aligned}
&\mathit{Guess}\ N^{(1)};\\
&\mathit{FAIL} = \mathit{failGeqSize}^{(1)}(N, K) \cup\\
&\qquad \pi_{\$1}\Big[(N \times N) \underset{\substack{\$1=EDGES.from\\ \wedge\\ \$2=EDGES.to}}{\bowtie} EDGES\Big].
\end{aligned}$$

The first subexpression of FAIL specifies the constraint $|N| \geq k$ (cf. Appendix G.1 for its definition), while the second one evaluates to the empty relation if and only if no pair of nodes in N is linked by an edge. An extension of N is an independent set (with size at least k) of the input graph if and only if the corresponding FAIL relation is empty.

Example 8.3.3 (Clique of a graph (62, Prob. GT19, p. 194)). *Given a graph as input and an integer $k \leq |NODES|$ encoded by a relation $K^{(1)}$ containing exactly k tuples, this* NP*-complete problem amounts to decide whether there exists a subset N of NODES, with $|N| \geq k$ such that every pair of distinct nodes of N is linked by an edge (i.e., the subgraph induced by N is* complete*).*

This problem can be specified in NP-Alg as follows:

$$\begin{aligned}&\mathit{Guess}\ N^{(1)};\\ &\mathit{FAIL} = \mathit{failGeqSize}(N,K) \cup \Big(\underset{\$1\neq\$2}{\sigma}(N \times N) \underset{\substack{\$1=\$1\\ \wedge\\ \$2=\$2}}{\bowtie} \mathit{complement}^{(2)}(\mathit{EDGES})\Big).\end{aligned}$$

The structure of the query is very similar to the one of the previous example, except for the new expression $complement^{(k)}(R^{(k)})$, that returns the active complement *(i.e., the complement with respect to the active domain of the database, cf. e.g., (2)) of the relation given as argument. Obviously the above query can be written in several other ways. As an example, a more efficient one would use the difference operator, instead of the join; notwithstanding this, we have chosen the above query to show the use of complement.*

The above examples show how *NP-Alg* is strictly related to ESO (cf. Section 1.2). In particular, relations in the set $\vec{Q}$ defined by the *Guess* construct play the same role of guessed predicates in an ESO specification of the form $\exists \vec{S}\ \phi(\vec{S}, \vec{R})$, while the expression for *FAIL* has the same role of $\phi(\vec{S}, \vec{R})$. Extensions for relations in $\vec{Q}$ encode points in the search space; a particular extension of guessed relations is a solution of the specified problem if and only if the expression for *FAIL* evaluates to $\emptyset$ (a discussion of the relationship between ESO and *NP-Alg* is presented in Section 8.4).

We can specify in *NP-Alg* other famous problems over graphs like *Dominating set*, *Transitive closure*, and *Hamiltonian path*. We remind that Transitive closure, indeed a polynomial-time problem, is not expressible in relational algebra (cf., e.g., (2)), because it intrinsically requires a form of recursion. In *NP-Alg* recursion can be simulated by means of guessing, as is for ESO. Hamiltonian path (HP) is the problem of finding a traversal of a graph that touches each node exactly once (cf. Example 3.2.1). The possibility to specify HP in *NP-Alg* leads to the possibility of having bounded integers in *NP-Alg*, and arithmetic operations on them, as it happens for ESO (cf. Section 1.2).

Other interesting problems, not involving graphs, can be specified in *NP-Alg*: *Satisfiability of a propositional formula* and *Evenness of the cardinality of a relation* are some examples.

Appendix G.1 describes useful syntactic sugar that can be introduced in *NP-Alg* without altering its expressive power.

8.4 Computational aspects of *NP-Alg*

In this section we focus on the main computational aspects of *NP-Alg*: data and combined complexity, expressive power, and polynomial fragments.

Technically, the results presented in this section can be easily obtained from corresponding ones formulated for other languages, e.g., ESO. Nevertheless, we believe that when designing a constraint modelling language it is of fundamental importance, from the methodological point of view, to ascertain its main computational properties.

8.4.1 Data and combined complexity

The *data complexity*, i.e., the complexity of query answering assuming the database as input and a fixed query (cf. (2)), is one of the most important computational aspects of a language, since queries are typically small compared to the database.

Since we can express NP-complete problems in *NP-Alg* (cf. Section 8.3), the problem of deciding whether $\mathit{FAIL} \diamond \emptyset$ is NP-hard. Moreover we can prove that the data complexity for such a problem is in NP by using the following argument. It is possible to generate, in non-deterministic polynomial time, an extension ext of Q. The answer is "yes" if and only if there is such an ext such that $\mathit{FAIL} = \emptyset$. The last check, being the evaluation of an ordinary relational algebra expression, can be done in polynomial time in the size of the database. The above considerations give us the first computational result on *NP-Alg*.

Theorem 8.4.1. *The data complexity of deciding whether $\mathit{FAIL} \diamond \emptyset$ for an NP-Alg query, where the input is the database, is* NP*-complete.*

Another interesting measure is *combined complexity*, where both the database and the query are part of the input. It is possible to show that, in this case, to determine whether $\mathit{FAIL} \diamond \emptyset$ is hard for the complexity class NE defined as $\bigcup_{c>1} \mathit{NTIME}\ (2^{cn})$ (cf. (105)), i.e., the class of all problems solvable by a non-deterministic machine in time bounded by 2^{cn}, where n is the size of the input and c is an arbitrary constant.

Theorem 8.4.2. *The combined complexity of deciding whether $\mathit{FAIL} \diamond \emptyset$ for an NP-Alg query, where the input is both the database and the query, is NE-hard.*

The proof is delayed to Appendix G.2.1.

8.4.2 Expressive power

The *expressiveness* of a query language characterizes the problems that can be expressed as fixed, i.e., instance independent, queries. In this section we prove the main result about the expressiveness of *NP-Alg*, by showing that it captures exactly NP, or equivalently (cf. (51)) queries in the existential fragment of second-order logic (ESO).

Of course it is very important to be assured that we can express *all* problems in the complexity class NP. In fact, Theorem 8.4.1 says that we are able to express *some* problems in this class. We remind that the expressive power of a language is in general less than or equal to its data complexity. In other words, there exist languages whose data complexity is hard for class C in which not every query in C can be expressed; several such languages are known, cf., e.g., (2).

In the following, we illustrate a method that transforms a formula in ESO (cf. Section 1.2) into an *NP-Alg* query. In particular, we deal with ESO formulae of the following kind:

$$\exists\vec{S}\ \forall\vec{X}\ \exists\vec{Y}\ \varphi(\vec{X},\vec{Y}), \tag{8.7}$$

where φ is a first-order formula (without quantifiers) containing variables among $\vec{X},\vec{Y}$ and involving relational symbols in $\vec{S}\cup\vec{R}\cup\{=\}$. The reason why we can restrict our attention to second-order formulae in the above normal form is explained in (82). As usual, "=" is always interpreted as "identity".

The transformation of a formula of the kind (8.7) into an *NP-Alg* expression ψ works in two steps:

1. The first-order formula $\varphi(\vec{X},\vec{Y})$ obtained by eliminating all quantifiers from (8.7) is translated into an expression *PHI* of plain relational algebra;

2. The overall *NP-Alg* query ψ is defined as:

$$\begin{aligned} &\mathit{Guess}\ Q_1^{(a_1)},\ldots,Q_n^{(a_n)};\\ &\mathit{FAIL} = \mathit{DOM}^{|\vec{X}|} - \pi_{\vec{X}}(\mathit{PHI}), \end{aligned} \tag{8.8}$$

where $a_1,\ldots,a_n$ are the arities of the n predicates in $\vec{S}$, and $|\vec{X}|$ is the number of variables occurring in $\vec{X}$.

The first step is rather standard (cf., e.g., (2)), and is briefly sketched here just to give the intuition. A relation R (with the same arity) is introduced for each predicate symbol r in the relational vocabulary of φ, i.e., $\vec{R}\cup\vec{S}$. A (function-free) atomic formula of first-order logic is translated as the corresponding relation, possibly prefixed by a selection that accounts for constant symbols and/or repeated variables, and by a renaming of attributes mapping the arguments. Selection can be used also for dealing with atoms involving equality. Inductively, the relation corresponding to a complex first-order formula is built as follows:

- $f\wedge g$ translates into $F\bowtie G$, where F and G are the translations of f and g, respectively;
- $f\vee g$ translates into $F'\cup G'$, where F' and G' are derived from the translations F and G to account for the (possibly) different schemata of f and g;
- $\neg f(\vec{Z})$ translates into $\rho_{\substack{\$1\to F.\$1\\ \vdots \\ \$|\vec{Z}|\to F.\$|\vec{Z}|}}(\mathit{DOM}^{|\vec{Z}|}-F)$.

It is worth noting that a better translation avoids the insertion of occurrences of the *DOM* relation for the important class of *safe formulae* (cf., e.g., (2)). However, these issues are out of the scope of this chapter, and will not be taken into account.

Relations obtained through such a translation will be called *q-free*, because they do not contain the projection operator (that plays the role of an existential *quantification*), except those implicit in equi-joins. Intuitively, this means that there are no existential quantifiers.

The following theorem claims that the above translation is correct.

Theorem 8.4.3. *For any* NP*-recognizable collection* $\vec{D}$ *of finite databases over* $\vec{R}$ *–characterized by a formula of the kind* (8.7)*– a database* D *is in* $\vec{D}$*, i.e.,*

$$D \models \exists \vec{S}\ \forall \vec{X}\ \exists \vec{Y}\ \varphi(\vec{X}, \vec{Y}),$$

if and only if FAIL$\Diamond\emptyset$*, when* ψ *(cf. formula* (8.8)*) is evaluated on* D*.*

The proof is delayed to Appendix G.2.

8.4.3 Polynomial fragments

Polynomial fragments of second-order logic have been presented in, e.g., (69). In this section we use some of those results to show that it is possible to isolate polynomial fragments of *NP-Alg*. In particular, the classes of *NP-Alg* queries identified by the following theorems correspond respectively to the Eaa and E_1e^*aa prefix classes of second-order logic described in (69), that are proved to be polynomial by a mapping into instances of 2SAT. The correctness of the translation is formally guaranteed by Theorem 8.4.3.

Theorem 8.4.4. *Let* s *be a positive integer, PHI a q-free expression of relational algebra over the relational vocabulary* $\vec{R} \cup \{Q^{(s)}\}$*, and* Y_1, Y_2 *the names of two attributes of PHI. An NP-Alg query of the form:*

$$\begin{aligned} &\mathit{Guess}\ Q^{(s)};\\ &\mathit{FAIL}\ = (\mathit{DOM} \times \mathit{DOM}) - \underset{Y_1,Y_2}{\pi} (\mathit{PHI}). \end{aligned}$$

can be evaluated in polynomial time in the size of the database.

Some interesting queries obeying the above restriction can indeed be formulated. As an example, *2-coloring* can be specified as follows (when $k = 2$, k-coloring, cf. Example 8.3.1, becomes polynomial):

$$\begin{aligned} &\mathit{Guess}\ C^{(1)};\\ &\mathit{FAIL} = \mathit{DOM} \times \mathit{DOM} - \left[\begin{array}{l} \mathit{complement}(\mathit{EDGES}) \cup \\ C \times \mathit{complement}(C)\ \cup\ \mathit{complement}(C) \times C \end{array}\right]. \end{aligned}$$

C and its complement denote the 2-partition. The constraint states that each edge must go from one subset to the other one.

Another polynomial problem of this class is *2-partition into cliques* (cf., e.g., (62)), that amounts to decide whether there is a 2-partition of the nodes of a graph such that the two induced subgraphs are complete. An *NP-Alg* query that specifies the problem is:

$$\begin{aligned}
&\textit{Guess } P^{(1)};\\
&\mathit{FAIL} = \mathit{DOM} \times \mathit{DOM} -\\
&\qquad [\mathit{complement}(P) \times P \;\cup\; P \times \mathit{complement}(P) \;\cup\; \mathit{EDGES}]\,.
\end{aligned}$$

A second polynomial class is defined by the following theorem.

Theorem 8.4.5. *Let* $\mathit{PHI}(X_1, \ldots, X_k, Y_1, Y_2)$ $(k > 0)$ *be a q-free expression of relational algebra over the relational vocabulary* $\vec{R} \cup \{Q^{(1)}\}$*. An NP-Alg query of the form:*

$$\begin{aligned}
&\textit{Guess } Q^{(1)};\\
&X(X_1, \ldots, X_k) = \mathit{PHI}(X_1, \ldots, X_k, Y_1, Y_2) \;/\; \underset{\substack{\$1 \to Y_1\\ \$2 \to Y_2}}{\rho} (\mathit{DOM} \times \mathit{DOM});\\
&\mathit{FAIL} \;= \mathit{empty}(X);
\end{aligned}$$

can be evaluated in polynomial time in the size of the database.

As an example, the specification for the *Graph disconnectivity* problem, i.e., to check whether a graph is not connected, belongs to this class.

8.5 The ConSql language

In this section we describe the ConSql language, a non-deterministic extension of SQL whose optimization-free subset has the same expressive power as *NP-Alg*, and present some specifications written in this language.

8.5.1 Syntax of ConSql

ConSql is a strict superset of SQL. The problem instance is described as a set of ordinary tables, using the data definition language of SQL. The novel construct `CREATE SPECIFICATION` is used to define a problem specification. It has three parts, two of which correspond to the parts of Definition 8.2.1:

1. Definition of the guessed tables, by means of the new keyword `GUESS`;
2. Definition of the objective function, by means of one of the two keywords `MAXIMIZE` and `MINIMIZE`;
3. Specification of the constraints that must be satisfied by guessed tables, by means of the standard SQL keyword `CHECK`.

Furthermore, the user can specify the desired output by means of the new keyword `RETURN`. In particular, the output is computed when an extension of the guessed tables satisfying all constraints and such that the objective function is optimized is found. Of course, it is possible to specify many guessed tables, constraints and returned tables. The syntax is as follows (we write it in BNF, with terminals either capitalized or quoted, and, for every terminal or non-terminal a, "[a]" meaning optionality, "$a*$" a list of an arbitrary number of a, and "$a+$" meaning "$a(a*)$"):

```
CREATE SPECIFICATION problem_name '('
       (GUESS TABLE table_name ['('aliases')'] AS guessed_table_spec)+
       ((MAXIMIZE | MINIMIZE) ' ('aggregate_query')'
       (CHECK '(' condition ')')+
       (RETURN TABLE return_table_name AS query)*
')'
```

The guessed table `table_name` gets its schema from its definition `guessed_table_spec`. The latter expression is similar to a standard `SELECT-FROM-WHERE` SQL query, except for the `FROM` clause that can contain also expressions such as:

```
SUBSET OF SQL_from_clause |
[TOTAL | PARTIAL] FUNCTION_TO '(' (range_table | min '..' max) ')'
      AS field_name_list OF SQL_from_clause |
(PARTITION '(' n ')' | PERMUTATION) AS field_name OF SQL_from_clause
```

with `SQL_from_clause` being the content of an ordinary SQL `FROM` clause (e.g., a list of tables). The schema of such expressions consists in the attributes of `SQL_from_clause`, plus the extra `field_name` (or `field_name_list`), if present.

In the `FROM` clause the user is supposed to specify the shape of the search space, either as a plain subset (like in *NP-Alg*), or as a mapping (i.e., partition, permutation, or function) from the domain defined by `SQL_from_clause`. Mappings require the specification of the range and the name of the extra field(s) containing range values. As for `PERMUTATION`, the range is implicitly defined to be a subset of integers. As for `FUNCTION_TO` the range can be either an interval `min..max` of a SQL enumerable type, (e.g., integers) or the set of values of the primary key of a table denoted by `range_table`. The optional keyword `PARTIAL` means that the function can be defined over a subset of the domain (the default is `TOTAL`). We remind that using partitions, permutations or functions does not add any expressive power to the language (cf. Appendix G.1.)

As for the objective function, the user is supposed to specify a query whose output is a monadic table with only one tuple of an SQL totally ordered type (e.g., integers or reals), typically by making use of SQL aggregate operators like `COUNT`, `SUM`, etc.

It is possible to specify constraints on the guessed tables by using ordinary SQL boolean conditions, e.g., `EXISTS`, `NOT EXISTS`, `IN`, `NOT IN`, `=ANY`, `=ALL`, etc.

Finally, the `query` that defines a returned table is an ordinary SQL query on the tables defining the problem instance plus the guessed ones, and it is evaluated for an extension of the guessed tables encoding an optimal solution.

Once a problem has been specified, its solution can be obtained with an ordinary SQL query on the return tables:

```
SELECT field_name_list
FROM problem_name.return_table_name
WHERE condition
```

The table `ANSWER(n INTEGER)` is implicitly defined locally to the `CREATE SPECIFICATION` construct, and it is empty if and only if the problem has no solution.

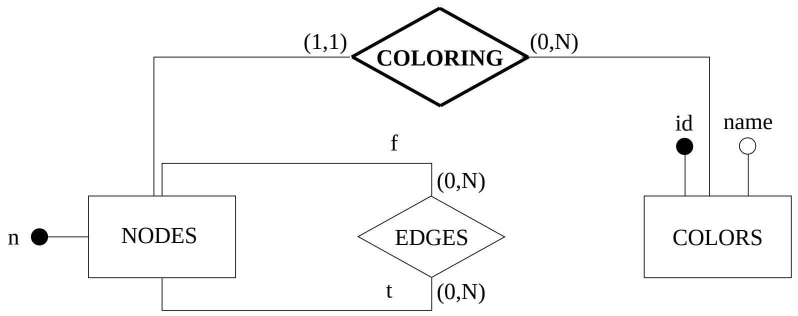

Figure 8.1: ER diagram of the database schema for the k-coloring problem. The guessed table *COLORING* is in boldface.

8.5.2 Examples

In this subsection we exhibit the specification of some problems in ConSql. In particular, to highlight its similarity with *NP-Alg*, we show the specification of the graph coloring problem of Example 8.3.1. Afterwards, we exploit the full power of the language and show how some real-world problems can be easily specified. In all the examples, we describe the schema of the input database, and underline key fields.

Graph k-coloring

We assume an input database over the schema shown in the Entity-Relationship (ER) diagram in Figure 8.1, thus containing relations *NODES(n)*, *EDGES(f,t)* (encoding the graph), and *COLORS(id,name)* (listing the k colors). Once a database (i.e., a problem instance) has been created (by using standard SQL commands), a ConSql specification of the k-coloring problem is the following:

```
CREATE SPECIFICATION Graph_Coloring (
   /* COLORING contains tuples of the kind <NODES.n, COLORS.id>,
      with COLORS.id arbitrarily chosen. */
 GUESS TABLE COLORING AS
   SELECT n, color FROM TOTAL FUNCTION_TO(COLORS) AS color OF NODES
    CHECK ( NOT EXISTS (
      SELECT * FROM COLORING C1, COLORING C2, EDGES
      WHERE C1.n <> C2.n AND C1.color = C2.color
        AND C1.n = EDGES.f AND C2.n = EDGES.t ))
    RETURN TABLE SOLUTION AS SELECT COLORING.n, COLORS.name
      FROM COLORING, COLORS WHERE COLORING.color = COLORS.id
)
```

The `GUESS` part of the problem specification defines a new (binary) table `COLORING`, with fields `n` and `color`, as a total function from the set of `NODES` to the set of `COLORS`. The `CHECK` statement expresses the constraint an extension of `COLORING` table must

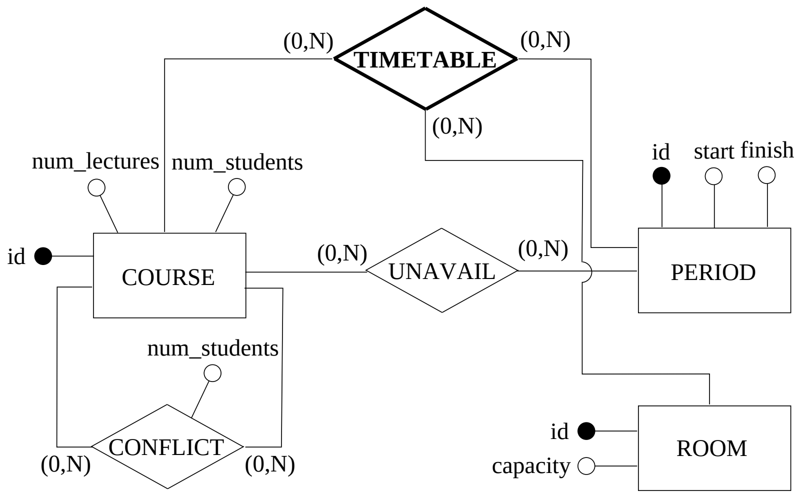

Figure 8.2: ER diagram of the database schema for the University course timetabling problem. The guessed table *TIMETABLE* is in boldface.

satisfy to be a solution of the problem, i.e., there are no two distinct nodes linked by an edge which are assigned the same color.

The `RETURN` statement defines the output of the problem by a query that is evaluated for an extension of the guessed table that satisfies every constraint. The user can ask for such a solution with the statement

```
SELECT * FROM Graph_Coloring.SOLUTION
```

As described in the previous subsection, if no coloring exists, the system table `Graph_Coloring.ANSWER` will contain no tuples. This can be easily checked by the user, in order to obtain only a significant `Graph_Coloring.SOLUTION` table.

University course timetabling

The *University course timetabling* problem (112) consists in finding the weekly schedule for all the lectures of a set of university courses in a given set of classrooms. We consider a variant of the original problem in which the objective function to minimize is the total number of students that wish to attend overlapping lectures.

The input database schema is shown in Figure 8.2, and consists of the following relations:

- *COURSE(<u>id</u>, num_lectures, num_students)*, consisting of tuples $\langle c, l, s\rangle$ meaning that the course c needs l lectures a week, and has s enrolled students.
- *PERIOD(<u>id</u>, start, finish)* encoding (non-overlapping) periods, plus information on start and finish time.

- *ROOM(id, capacity)*. A tuple $\langle r, c\rangle$ means that room r has capacity c.
- *CONFLICT(course1, course2, num_students)*. A tuple $\langle c1, c2, n\rangle$ means that courses *c1* and *c2* have n common students.
- *UNAVAILABLE(course, period)*. A tuple $\langle c, p\rangle$ means that the teacher of course c is not available for teaching at period p.

A solution to the problem is a (guessed) relation *TIMETABLE(period, room, course)* with tuples $\langle p, r, c\rangle$ meaning that at period p in room r there is a lecture of course c. If for some values of the *room* and *period* fields there is no tuple in the relation *TIMETABLE*, then the room is unused in that period.

A ConSql specification of the timetabling problem, given an input database, is the following:

```
CREATE SPECIFICATION University_Timetabling (
  GUESS TABLE TIMETABLE(period, room, course) AS
    SELECT p.id, r.id, course
    FROM PARTIAL FUNCTION_TO(COURSE) AS course OF PERIOD p, ROOM r
  // Objective function
  MINIMIZE ( SELECT ( SUM(c.num_students)
    FROM TIMETABLE t1, TIMETABLE t2, CONFLICT c
    WHERE t1.period = t2.period AND t1.course <> t2.course AND
          c.course1 = t1.course AND c.course2 = t2.course
  ))
  // At most one lecture of a course per period
  CHECK ( NOT EXISTS (
    SELECT * FROM TIMETABLE t1, TIMETABLE t2
    WHERE t1.course = t2.course AND
          t1.period = t2.period AND t1.room <> t2.room
  ))
  // Unavailability constraints
  CHECK ( NOT EXISTS (
    SELECT * FROM TIMETABLE t, UNAVAILABLE u
    WHERE t.course = u.course AND t.period = u.period
  ))
  // Capacity constraints
  CHECK ( NOT EXISTS (
    SELECT * FROM TIMETABLE t, COURSE c, ROOM r
    WHERE t.course = c.id AND t.room = r.id AND
          c.num_students > r.capacity
  ))
  // Teaching requirements
  CHECK ( NOT EXISTS (
    SELECT * FROM COURSE c
    WHERE c.num_lectures <>
      ( SELECT COUNT(*) FROM TIMETABLE t
          WHERE t.course = c.id
      )
  ))
  RETURN TABLE SOLUTION AS SELECT * FROM TIMETABLE
)
```

In particular, the constraints force extensions of the guessed table *TIMETABLE* to be such that:

- There is at most one lecture of a course per period, i.e., there cannot be two different rooms allocated for the same course in the same time slot;
- Unavailability constraints are respected, i.e., no lecture is scheduled in a period for which the relevant teacher is unavailable;
- Capacity constraints are respected, i.e., no room is allocated for courses having a number of students that exceeds its capacity;
- Teaching requirements are satisfied, i.e., all courses have a room and a time slot assigned for all the lectures they need.

An extension for guessed table *TIMETABLE* that satisfies the constraints above is an optimal solution of the University course timetabling problem if it minimizes the overall number of students that are expected to attend conflicting lectures, i.e., lectures that are scheduled at the same time.

Aircraft landing

The *aircraft landing* problem (6) consists in scheduling landing times for aircraft. Upon entering within the radar range of the air traffic control (ATC) at an airport, a plane requires a *landing time* and a *runway* on which to land. The landing time must lie within a specified time window, bounded by an *earliest time* and a *latest time*, depending on the kind of the aircraft. Each plane has a most economical, preferred speed. A plane is said to be assigned its *target time*, if it is required to fly in to land at its preferred speed. If ATC requires the plane to either slow down or speed up, a cost incurs. The bigger the difference between the assigned landing time and the target landing time, the bigger the cost. Moreover, the amount of time between two landings must be greater than a specified minimum (the *separation time*) that depends on the planes involved. Separation times depend on the aircraft landing on the same or different runways (in the latter case they are smaller).

Our objective is to find a landing time for each planned aircraft, encoded in a guessed relation *LANDING*, satisfying all the previous constraints, and such that the total cost (i.e., the sum of the costs associated with each aircraft) is minimized. The input database schema is shown in Figure 8.3, and consists of the following relations:

- $AIRCRAFT(\underline{id}, target\_time, earliest\_time, latest\_time, bef\_cost, aft\_cost)$, listing aircraft planned to land, together with their target times and landing time windows; the cost associated with a delayed or advanced landing at time x is given by $bef\_cost \cdot \max[0, t-x] + aft\_cost \cdot \max[0, x-t]$, where t is the aircraft target time.
- $RUNWAY(\underline{id})$ listing all the runways of the airport.

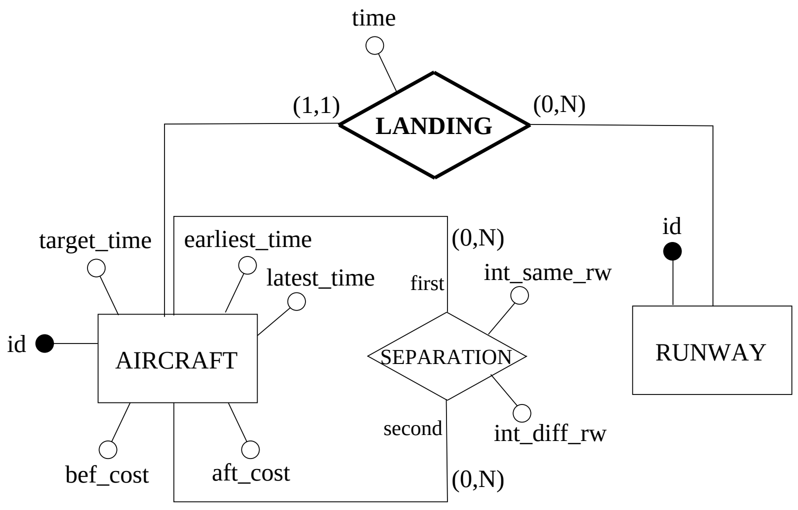

Figure 8.3: ER diagram of the database schema for the Aircraft landing problem. The guessed table *LANDING* is in boldface.

- *SEPARATION*(<u>*first*</u>, <u>*second*</u>, *int_same_rw*, *int_diff_rw*). A tuple $\langle a, a', is, id\rangle$ means that if aircraft a' lands after aircraft a, then landing times must be separated by *is* (resp. *id*) minutes if they land on the same runway (resp. on different runways).

In the following specification, the search space is a total function assigning an aircraft to a landing time and a runway. For the sake of simplicity, landing times are expressed in minutes after a conventional time instant, e.g., the scheduling starting time, and the time horizon is set to one day, i.e., 24×60 minutes.

```
CREATE SPECIFICATION Aircraft_Landing (
  GUESS TABLE LANDING AS
    SELECT ar.id, ar.runway, at.time
    FROM (TOTAL FUNCTION_TO(RUNWAY) AS runway OF AIRCRAFT) ar,
         (TOTAL FUNCTION_TO(0..24*60-1) AS time OF AIRCRAFT) at
    WHERE ar.id = at.id
   // Objective function
   MINIMIZE ( SELECT SUM(cost)
     FROM (
       SELECT a.id, (a.bef_cost * (a.target_time - l.time)) AS cost
         FROM AIRCRAFT a, LANDING l
         WHERE a.id = l.aircraft AND l.time <= a.target_time
       UNION // advanced plus delayed aircraft
       SELECT a.id, (a.aft_cost * (l.time - a.target_time)) AS cost
         FROM AIRCRAFT a, LANDING l
         WHERE a.id = l.aircraft AND l.time > a.target_time
     ) AIRCRAFT_COST // Contains tuples <aircraft, cost>
   )
   // Time window constraints
   CHECK ( NOT EXISTS (
     SELECT * FROM LANDING l, AIRCRAFT a WHERE l.aircraft = a.id
       AND ( l.time > a.latest_time OR l.time < a.earliest_time )
   ))
   // Separation constraints
   CHECK ( NOT EXISTS (
     SELECT * FROM LANDING l1, LANDING l2, SEPARATION sep
     WHERE l1.aircraft <> l2.aircraft AND l1.time <= l2.time AND
       sep.first = l1.aircraft AND sep.second = l2.aircraft AND
       (  ( (l1.runway = l2.runway) AND
            (l2.time - l1.time) < sep.int_same_rw ) OR
          ( (l1.runway <> l2.runway) AND
            (l2.time - l1.time) < sep.int_diff_rw )
   )))
   RETURN TABLE SOLUTION AS SELECT * FROM LANDING
)
```

In particular, the constraints force extensions of the guessed table *LANDING* to be such to respect both time window constraints (i.e., the actual landing time for each aircraft must lie inside its landing time window), and separation constraints (encoded in the *SEPARATION* relation). Such an extension is an optimal solution of the Aircraft landing problem if it minimizes the overall cost.

8.5.3 ConSql simulator

ConSql simulator is an application that works as an interface to a traditional R-DBMS. It simulates the behavior of a ConSql server by reading from its input stream ConSql queries, i.e., ordinary sql queries and commands, and problem specifications. Ordinary sql queries and commands are simply passed to the underlying R-DBMS, while problem specifications are processed. The overall architecture of the system is depicted in Figure 8.4. In particular, `CREATE SPECIFICATION` constructs are parsed, creating the new tables (corresponding to the guessed ones) and an internal representation of the search space. The search space is then explored by the solver, looking for an element corresponding to an optimal solution, by posing appropriate queries to the R-DBMS (in standard sql). As soon as the optimal solution is found, the results of the query specified in the `RETURN` statements are accessible to the user as output.

The implementation of ConSql simulator gives much attention to software engineering aspects and to different quality factors of software artifacts. In particular, the system is platform independent and highly portable, since it is written in Java, and uses the standard JDBC protocol for the connection with the R-DBMS, and the whole architecture presents a neat separation among the language parser, the problem modelling module and the solver engine (jLocal), so as to emphasize qualities such as modularity, extendability and reusability. In particular, the problem modelling module allows to represent problem specifications in a *language independent* fashion, relying on abstract concepts such as *Problem*, *Search space*, *Objective function*, and

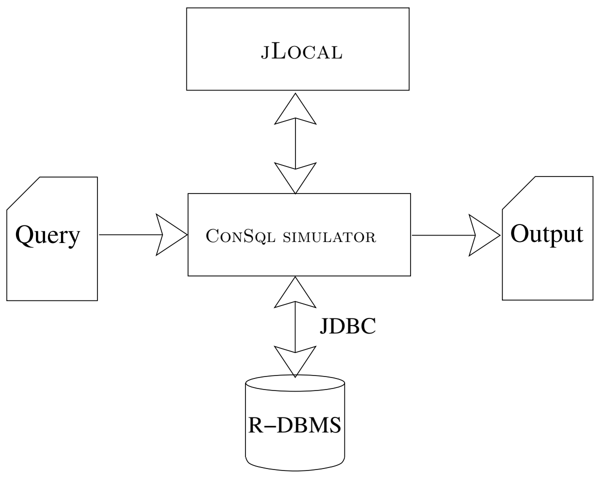

Figure 8.4: ConSql simulator overall architecture.

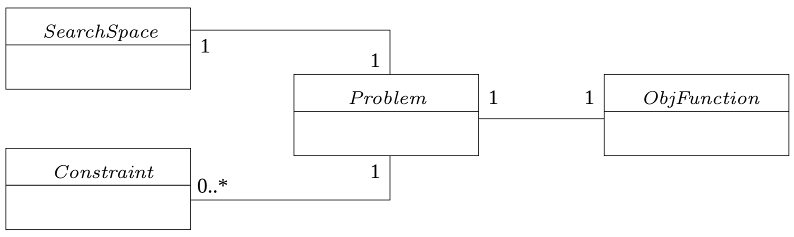

Figure 8.5: Portion of the conceptual UML class diagram for the language-independent problem modelling module.

Constraint, as the conceptual UML diagram in Figure 8.5 shows. In this way, the solving engine jLocal, interacting with the abstract problem modelling module, is independent of the particular language, i.e., ConSql. The only language-dependent part of the system is the parser for the ConSql language, that provides concrete implementations for the abstract concepts that compose the problem modelling module, building the internal representation of the problem instance, the search space, and the constraints, and for some of the services needed by the solver.

As for the search methods, the solver engine jLocal exploits local search techniques to find solutions. Local search is considered one of the most attractive techniques for solving combinatorial optimization problems (cf., e.g., (1)), being able to solve instances of realistic size in reasonable time. Several local search algorithms have been implemented in jLocal, among them Hill climbing and Tabu search (67), and several strategies that combine different solvers for doing the search are present (e.g., Tandem search, in which two different solvers are used in sequence). Additional local search strategies can be simply added by subclassing the *LocalSearchSolver* class (not described here for the sake of simplicity).

It is worth noting that the user is completely unaware of the search techniques implemented by the system. In particular, definitions for neighborhoods, moves, aspiration functions, etc., are made by ConSql simulator itself, starting from the types of the guessed tables defined in the specification (i.e., subsets, functions, permutations, partitions).

The development of ConSql simulator has been done according to the iterative model of the software life-cycle. In particular, three iterations were expected. The first, prototypical, version of the system, which was used only to test specifications, relied on a purely enumerative approach, and, of course, no considerations on performances could be done. As for the present version, which is at the second iteration of the development process, we added the local search engine, but the connection with the DBMS is yet completely black box. In particular, jLocal uses the DBMS both for maintaining the current state, for checking constraints, and for evaluating which neighbor to visit next. The main motivation behind this choice, is that current DBMSs offer means to answer queries efficiently, especially in case of very large instances. Nonetheless, since constraints are evaluated from scratch in every visited state, performances cannot be good, and only instances of small sizes can be actually solved.

In the third version of the system, which is currently under development, we are adding the following functionalities:

1. The ability of checking constraints incrementally: constraints' check is the main source of inefficiency of the current version, since they are evaluated from scratch for every visited state, and for all its neighbors, in order to choose the best move. Hence, the number of queries posted to the DBMS is very high, and all of them are answered independently from each other. However, it is clear that, when using local search technology, only a small variation in the number of constraint violations is expected, when moving from one state to its neighbors. To this end, our goal is to make the DBMS able to compute only variations to constraints' violations when performing checks. This is expected to greatly increase the overall performances of the system, since we can rely on very sophisticated algorithms to, e.g., maintain and synchronize views, actually present in currently available DBMSs.

2. The use of a much more complex local search engine. In particular, we are currently integrating EasyLocal++ (45), a very sophisticated solver, with our system. This can lead to better algorithms, and to a fine tuning of their parameters.

3. The addition of an optional "search" part in `CREATE SPECIFICATION` constructs, as it happens in, e.g., opl, in order to provide the user with the possibility of declaring which search algorithm to adopt, as well as the types of neighborhoods, moves, aspiration functions, etc. Of course, as already claimed, our goal is to provide good defaults for the options in this part, as, e.g., opl does, by letting the system able to automatically make a good choice for these issues, depending on the specification at hand.

4. To provide a better coupling with a particular open-source DBMS, in order to make the system able to directly use the DBMS' APIs, instead of interacting by means of (inefficient, but highly portable) protocols like JDBC.

8.6 Discussion, related work and future research directions

In this chapter we have tackled the issue of strong integration between constraint modelling and programming and up-to-date technology for storing data. In particular we have proposed constraint languages that have the ability to interact with data repositories in a standard way. To this end, we have presented *NP-Alg*, an extension of relational algebra which is specially suited for combinatorial problems. The main feature of *NP-Alg* is the possibility of specifying, via a form of non-determinism, a set of relations that can have an arbitrary extension. This allows the specification of a search space suitable for the solution of combinatorial problems, with ordinary relational algebra expressions defining constraints. Although *NP-Alg* provides just a very simple guessing operator, many useful search spaces, e.g., permutations and functions, can be defined as syntactic sugar.

Several computational properties of *NP-Alg* have been shown, including data and combined complexity, and expressive power. Notably, the language is shown to capture exactly all the problems in the complexity class NP, which includes many combinatorial problems of industrial relevance. In the same way, we have proposed ConSql, a non-deterministic extension of sql, with the same expressive power of *NP-Alg*, which is suitable also for specifying optimization problems. The effectiveness of *NP-Alg* and ConSql both as complex query and constraint modelling languages has been demonstrated by showing several queries that specify combinatorial problems.

Other extensions of relational algebra have already been proposed. The most important examples are the languages *Alg* + *while* and *Alg* + *while*$^+$, where, respectively, a non-inflationary and an inflationary fixpoint semantics is added (2). Both these languages are capable of expressing the Transitive Closure query, but have very different expressive power: *Alg* + *while* can express queries in PSPACE, but the language captures exactly this class only on ordered databases (i.e., databases in which a total order among all constants occurring in it is fixed). As for *Alg* + *while*$^+$ instead, it can express only polynomial-time queries, and the language captures the whole PTIME class only on ordered databases. A feature for expressing linear recursion has recently been added also to sql (sql'99), by means of the `WITH` construct. However, both the aforementioned extensions of relational algebra, and the new version of sql do not make such languages suitable for expressing constraint problems.

Several languages and systems for constraint programming are nowadays available either as research and commercial packages. Some of them are in the form of frameworks and libraries. As an example, in ECL^iPS^e (49) or SICStus (117) a traditional programming language such as Prolog is enhanced by means of libraries and specific constructs for specifying constraints, which are then solved by highly optimized algorithms. The ILOG Optimization suite (73) provides instead libraries for expressing constraints callable by host general-purpose programming languages like C++.

Specification languages natively developed for constraint modelling and programming are also available, either commercially like or as research prototypes, and some of them have been already cited in Section 1.1. All of them offer an ad-hoc syntax for problem specifications, but differently from them, *NP-Alg* and ConSql use standard and well-known languages for specifying problem specifications, that are considered just like queries over a relational database representing the input instance. We believe that this feature permits a wider diffusion of the declarative constraint modelling paradigm in industrial environments, permitting a very strong integration with the information system of the enterprise. Conversely, the other systems usually get input data from text files in ad-hoc formats, and additional machinery is needed to build such files from the content of a relational database, and for storing problem solutions. Even if such machinery is sometimes offered by means of libraries and/or plug-ins (cf., e.g., (74)), relational data is always manipulated outside of the database, with a consequent potential lack of data integrity and other transactional properties. In fact, the most currently adopted solution for evaluating complex queries over a relational database is to embed SQL into external programs, written in a general-purpose programming language, like Java or C++.

Hence, we believe that ConSql is a clear step towards a language for both declarative constraint modelling and complex queries to relational databases, which relies on standard and well-known technologies. The language has been explicitly designed for being implemented inside the R-DBMS, so guaranteeing all transactional properties to the query evaluation process.

Several query languages capable of capturing the complexity class NP have been shown in the literature. As an example, in (82) an extension of datalog (the well-known recursive query language (123)) allowing negation is proved to have such a property. Another extension of datalog capturing NP, without negation but with a form of non-determinism, is proposed in (25). Other rule-based languages with different semantics have also been proposed: Smodels (118) which relies on stable models semantics, and DLV (85) which is based on answer set programming. They also are based on negation and recursion. On the other hand, *NP-Alg* captures NP without recursion. Actually, recursion can be simulated by non-determinism, and it is possible to write, e.g., the transitive closure query in *NP-Alg*. Being non-recursive, *NP-Alg* is more similar to plain second-order existential logic. Nevertheless, it retains the functional character of relational algebra, that sometimes makes it easier (with respect to rule-based languages) to specify a problem.

As for the proposed implementation of ConSql simulator, it is conceived to be based on a purely declarative language and to be ready to use, i.e., it does not require any additional code to be written by the user. Other systems for local search do, however, exist, either in forms of declarative languages for modelling in a concise way local search algorithms. (cf., e.g., (96; 127)) or, alternatively, in forms of libraries or frameworks (cf., e.g., Local++ (113)), hence providing algorithms that rely on additional application-specific code provided by the user. ConSql simulator is different from such systems in that it provides the user with the ability of modelling an optimization problem by means of a language, i.e., ConSql, that is completely unaware of the particular solving technology used. It is responsibility of the engine to provide the local search solver with all the information needed to explore the search space (e.g., description of neighborhoods, moves, etc.). This choice is currently made starting from the types of the guessed tables defined in the specification, and future work has to be done in order to better exploit the different alternatives.

As for the solving engine, we note that the strong separation of the language-independent solver jLocal from the ConSql parser permits an easy reuse of the solver with other problem specification languages. Actually, jLocal is yet in a prototypal stage, and more work is needed in order to extend and optimize the implemented search algorithms.

ConSql simulator and jLocal will be released as free and potentially open source software, thus allowing the packages to receive improvements and extensions from the community.

Results in this chapter have been published in (17; 16).

Chapter 9

Conclusions

In this thesis we focused on *declarative constraint modelling* of constraint problems, arguing that this paradigm is going to be the next challenge for Constraint Programming systems.

In fact, current languages and systems to be efficient, still require that the user provides a "good" model of the problem, and explicitly states some procedural aspects, like search procedures. This implies that problem modelling must be performed with taking procedural aspects in mind, hence leading to a high lowering in the degree of declarativeness. As a consequence, modelling has yet to be considered as an "art", and a strong expertise is requested to users in order to be able to solve real problems in practice.

From these observations, we started the investigation of different and complementary techniques to let the system able to perform automatically some kind of reasoning on the specifications given by the user, with the goal of reformulating them in order to improve the efficiency of the solving process.

Differently with all other approaches in the literature, all the proposed techniques apply at the symbolic level of the specification, and so regardless of the particular instance to be solved. This is coherent with the clear separation, exhibited by many state-of-the-art systems for CP, between problem specification and instance, and is similar, in a sense, to the approach used in relational databases in order to deal with the query optimization problem, that is known to rely on the query and the database schema only, and not on the current database content (cf. Section 1.1). In our vision, all "structural" properties of constraint problems that are amenable to be optimized should be recognized and exploited at the level of the specification, letting to the subsequent stage the task of detecting and exploiting only the additional properties that may arise from particular instances. On the other hand, existing techniques for problem optimization try to infer *all* properties after instantiation, when the structure of the problem has been almost completely hidden.

In order to be able to formally deal with problem specifications, we observed (cf. Section 1.2) that they can be regarded as logical formulae in existential second-order logic (ESO). To this end, all the proposed techniques for problem reformulation deal with ESO formulae, but of course, they can be (often straightforwardly) extended to richer languages for constraint modelling, like AMPL or OPL, as some examples show.

Chapters 2 to 5 deal with different techniques for reformulating problem specifications given as ESO formulae. In particular, in Chapters 2 and 3 we studied how to characterize those constraints whose evaluation can be safely delayed, in order to reduce the original problem to a simplified one, with a larger set of solutions, while in Chapters 4, 5, and 6 we investigated the problem of detecting symmetries and functional dependencies at the specification level, showed how automated tools, like first-order theorem provers and finite model finders, can be used in practice in order to automate the discovery process, and discussed how detected properties can be automatically exploited in order to save search. Experimental results at the end of each chapter evidence the effectiveness of the various approaches.

Chapter 7 deals with Knowledge Compilation (KC), a slightly different technique for dealing with intractable problems. In particular, we give sufficient syntactic conditions for a specification to be able to take advantage of compilation, in order to efficiently solve several instances that share a common part, and show how many compilable problems fit in this characterization. Moreover, we proved the reasonability of the presented classification, by showing that, by relaxing hypotheses, sets of specifications containing non-compilable ones are obtained.

In Chapter 8 we discuss two highly declarative languages for constraint modelling, *NP-Alg* and ConSql, which are non-deterministic extensions of relational algebra and SQL, respectively. The desiderability of such languages has been justified by observing how state-of-the-art systems do not provide a strong integration between constraint solving engines and standard data repositories, and that this can be an obstacle for a wider diffusion of the constraint modelling paradigm in industrial environments, where issues concerning data integrity are very important. Since *NP-Alg* and ConSql have been designed in order to extend standard query languages to relational databases, they can be completely embedded in next generation R-DBMSs, and problem specifications given in these languages can be regarded exactly as more complex queries to a relational database. We believe that this can be a starting point for a wider diffusion of CP in industrial environments, since, for the non-academic user to exploit CP, it will not be necessary to learn a completely new language and methodology. Computational properties of proposed languages have been investigated, e.g., data and combined complexity, as well as expressive power, and polynomial classes of queries have been isolated. In particular, the latter issue highlights how regarding problem specifications as logical formulae makes it possible to refer to existing results in second-order logic.

Appendix A

ESO encodings of expressions in Section 1.2

In this section we give the formal definitions for expressions given in Section 1.2, in order to let ESO deal with functions, integers, and orderings.

A.1 Expressions for functions

In what follows, $\vec{X}$, $\vec{X_1}$, and $\vec{X_2}$ are vectors of size d, while $\vec{Y}$, $\vec{Y_1}$, and $\vec{Y_2}$ are of size r. We also assume that we are given relations D of arity d and R of arity r.

Definitions of expressions of Section 1.2 are as follows:

- $\mathit{typedRelation}(F^{(d+r)}, D^{(d)}, R^{(r)}) \ \doteq$

$$\forall\vec{X}\ \forall\vec{Y}\ \ F(\vec{X},\vec{Y}) \ \rightarrow\ D(\vec{X}) \wedge R(\vec{Y}) \tag{A.1}$$

- $\mathit{function}_{d,r}(F^{(d+r)}) \ \doteq$

$$\forall\vec{X}\ \forall\vec{Y_1}\ \forall\vec{Y_2}\ F(\vec{X},\vec{Y_1}) \wedge F(\vec{X},\vec{Y_2}) \ \rightarrow\ \vec{Y_1} = \vec{Y_2} \tag{A.2}$$

- $\mathit{total}(F^{(d+r)}, D^{(d)}) \ \doteq$

$$\forall\vec{X}\ D(\vec{X}) \ \rightarrow\ \exists\vec{Y}\ \ F(\vec{X},\vec{Y}) \tag{A.3}$$

- $\mathit{surgective}(F^{(d+r)}, R^{(r)}) \ \doteq$

$$\forall\vec{Y}\ R(\vec{Y}) \ \rightarrow\ \exists\vec{X}\ \ F(\vec{X},\vec{Y}) \tag{A.4}$$

- $\mathit{injective}_{d,r}(F^{(d+r)}) \quad \doteq$

$$\forall \vec{Y}\ \forall \vec{X}_1\ \forall \vec{X}_2\ F(\vec{X}_1, \vec{Y}) \wedge F(\vec{X}_2, \vec{Y}) \ \rightarrow\ \vec{X}_1 = \vec{X}_2 \tag{A.5}$$

- $\mathit{bijective}(F^{(d+r)}, D^{(d)}, R^{(r)}) \quad \doteq$

$$\mathit{total}(F, D) \ \wedge\ \mathit{injective}(F) \ \wedge\ \mathit{surjective}(F, R) \tag{A.6}$$

A.2 Bounded integers and linear orders

The simplest way to represent bounded integers in ESO is to encode them in unary form. In particular, positive integer k can be represented by a (wlog monadic) relation K having exactly k tuples. Hence, given two monadic relations K' and K'' representing positive integers k' and k'', comparisons between them can be reduced to checking the existence of appropriate functions from K' to K''. As an example, we have that:

- $k' = k''$ if and only if there exists a total bijective function between K' and K'';
- $k' \geq k''$ if and only if there exists a partial surjective function from K' to K''.

However, an alternative representation of bounded integers is described below.

For what concerns orderings, we observe that in ESO it is possible to specify the NP-complete *Hamiltonian Path on a graph* problem, as the following example shows:

Example A.2.1 (Hamiltonian path (62, Prob. GT39, p. 199)). *Given a directed graph as input, encoded in a binary relation $edge(\cdot,\cdot)$, this* NP*-complete problem amounts to decide whether there exists a traversal of the input graph that touches every node exactly once. An ESO specification for the Hamiltonian path (HP) problem is as follows:*

$$\exists P, TC \ \ \forall X, Y\ P(X,Y) \ \rightarrow\ \mathit{edge}(X,Y) \wedge \tag{A.7}$$
$$\exists ! S, T\ \forall X\ \neg P(X,S) \ \wedge\ \neg P(T,X) \ \wedge \tag{A.8}$$
$$\forall X\ X \neq T \ \leftrightarrow\ \exists ! Y\ P(X,Y) \wedge \tag{A.9}$$
$$\forall Y\ Y \neq S \ \leftrightarrow\ \exists ! X\ P(X,Y) \wedge \tag{A.10}$$
$$\forall X\ \exists Y\ P(X,Y) \ \vee\ P(Y,X) \wedge \tag{A.11}$$
$$\mathit{closure}(P, TC) \wedge \tag{A.12}$$
$$\forall X\ \neg TC(x,x). \tag{A.13}$$

Constraint (A.7) *forces P to be a subset of the graph edges. Moreover, exactly two*[1] *distinguished nodes S and T exist in P: the former will be the source node of the Hamiltonian path, the latter the target one. Constraint* (A.8) *forces node S to have*

[1]Formula $\exists ! X\ \phi(X)$ means that exists exactly one X that makes $\phi(X)$ true, and is a common short-hand for $\exists X\ \phi(X)\ \wedge\ \forall X'\ \phi(X') \rightarrow X = X'$.

no predecessors and T to have no successors. Constraints (A.9–A.10) impose every node to have exactly one successor and one predecessor in P, except for S (which has no predecessor) and T (which has no successor). Finally, (A.11) *forces every node to be touched by the path encoded in P. Constraints (A.7–A.11) force the graph induced by P to be a set of one chain (starting from S and ending in T) plus zero or more cycles (sub-cycles), all disconnected from each other. An extension for P that satisfies (A.7–A.11) is an Hamiltonian path if and only if it has no such cycles. Additional constraints (A.12–A.13) check this: in particular, the formes ensures that the additional guessed predicate Cl is a* closure *of the graph induced by P (this can be the transitive closure, but simpler forms of closure may suffice, since the structure of the graph is known), while the latter checks that no tuple of the form $\langle X, X\rangle$ belongs to Cl: this is true if and only if the path encoded by P does not have cycles. A simple expression for closure$(\cdot,\cdot)$ is as follows:*

$$\begin{aligned} &\mathit{closure}(R,T) \doteq \\ &\qquad \forall X,Y\ T(X,Y) \leftrightarrow R(X,Y) \vee (\exists Z\ T(X,Z) \wedge T(Z,Y)), \end{aligned} \tag{A.14}$$

and evaluates to true if and only if binary relation T encodes this particular kind of closure of the binary relation R. It is easy to prove that guessed predicate Cl satisfying closure(P, Cl) contains a tuple $\langle X, X\rangle$ if and only if P (which has the structure imposed by the preceeding constraints) has a cycle.

A formulation for HP as permutation problem is given in Example 3.2.1.

The possibility to specify HP has interesting consequences. Consider a unary relation N, with $n \neq 0$ tuples, and the *complete* graph C with nodes in N (hence, having n nodes and n^2 edges). An Hamiltonian path H of C is a *total ordering* of the n elements in N: in fact it is a *successor* relation. The Transitive closure of H (also expressible in ESO by a formula more complex than (A.14), that intuitively simulates the behavior of the well-known datalog program for its computation) is the corresponding *less-than* relation. As a consequence, we have an alternative representation in ESO of *bounded integers* up to n (every tuple of N is associated an integer), and, furthermore, we can also specify arithmetic operations over them, as long as results of those operations is bounded.

Hence, expression $\mathit{linearOrder}(\mathit{ORDER}^{(2k)}, N^{(k)})$ actually checks whether guessed relation *ORDER* defines an HP over the complete graph with nodes in N.

Appendix B

Chapter 2: proofs of results

Proof of Theorem 2.2.1. Let $\mathcal{I}$ be an instance, M^* be a list of extensions for $(S_1, \ldots, S_h, S^*)$ such that $(M^*, \mathcal{I}) \models \psi^s$, and $ext(S)$ be an extension for S such that $(M^*, ext(S), \mathcal{I}) \models \psi^d$.

From the definition of ψ^d, it follows that:

$$(M^*, ext(S), \mathcal{I}) \models \forall X \ S(X) \rightarrow S^*(X),$$

and so, since clauses in Ξ^* contain at most negative occurrences of S^*, that:

$$(M^*, ext(S), \mathcal{I}) \models \Xi. \tag{B.1}$$

Furthermore, from the definition of ψ^s it follows that:

$$(M^*, \mathcal{I}) \models \forall X \ \alpha(X) \rightarrow S^*(X),$$

and from **Hyp 2** that:

$$(M^*, \mathcal{I}) \models \forall X \ \alpha(X) \rightarrow S^*(X) \wedge \beta(X).$$

This implies, by the definition of ψ^d, that:

$$(M^*, ext(S), \mathcal{I}) \models \forall X \ \alpha(X) \rightarrow S(X). \tag{B.2}$$

Moreover, by the same definition, it is also true that:

$$(M^*, ext(S), \mathcal{I}) \models \forall X \ S(X) \rightarrow \beta(X). \tag{B.3}$$

From (B.1–B.3), and from the observation that S^* does not occur in any of the right parts of them, the thesis follows. □

Proof of Theorem 2.2.2. Let $\mathcal{I}$ be an input instance, and M be a list of extensions for $(S_1, \ldots, S_h, S)$ such that $(M, \mathcal{I}) \models \psi$. Let S^* be defined in such a way that $ext(S^*) = ext(S)$.

Since solutions for ψ are also solutions for ψ^s, and since $ext(S^*) = ext(S)$, it follows that $((M - ext(S)) \cup ext(S^*))$ is a solution of ψ^s.

As for ψ^d, we observe that since M (that is a solution for the whole specification ψ) is also a solution for one of its constraints, namely $\forall X\ S(X) \rightarrow \beta(X)$ (the delayed constraint), then, from $ext(S^*) = ext(S)$, and in particular from the fact that $\forall X\ S(X) \rightarrow S^*(X)$ holds, it follows that $(M, ext(S^*)) \models \forall X\ S(X) \rightarrow S^*(X) \wedge \beta(X)$.

Conversely, from $ext(S^*) = ext(S)$, and in particular from the fact that $\forall X\ S^*(X) \rightarrow S(X)$ holds, it follows that $(M, ext(S^*)) \models \forall X\ S^*(X) \wedge \beta(X) \rightarrow S(X)$.

Hence, the thesis follows. □

Appendix C

Chapter 4: proofs of results and examples

Proof of Proposition 4.3.1.
If part. If $\phi \equiv \phi^{\sigma}$, then it is easy to show that σ meets the condition of Definition 4.2.5, thus it is a UVS for ψ.
Only if part. If σ is a UVS for ψ, then, according to Definition 4.2.5, every extension for predicates in $\vec{S}$ and $\vec{R}$ that is a model of ϕ must be a model of ϕ^{σ} and vice versa, therefore $\phi \equiv \phi^{\sigma}$ must hold. ☐

Proof of Proposition 4.3.2. We prove the statement by reducing it to the problem of checking whether an arbitrary closed first-order formula is a contradiction.

Let $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$ be any fixed specification on input schema $\vec{R}$, and let σ be any fixed UVT which is *not* a UVS for it, i.e., from Proposition 4.3.1:

$$\phi \not\equiv \phi^{\sigma}. \tag{C.1}$$

We remind that ϕ^{σ} is defined as $\phi[S_1/\sigma(S_1), \ldots, S_n/\sigma(S_n)]$ (cf. Section 4.2).
Let now $\vec{P}$ be a fresh set of relational symbols, with $\vec{P} \cap \{\vec{S} \cup \vec{R}\} = \emptyset$, $\gamma(\vec{P})$ an arbitrary closed first-order formula on the relational vocabulary $\vec{P}$, and consider the following new specification ψ':

$$\exists \vec{S}\ \phi(\vec{S}, \vec{R}) \wedge \gamma(\vec{P}).$$

The UVT σ is also a UVT for ψ', since ψ' has the same set of guessed predicates as ψ. By Proposition 4.3.1, the problem of deciding whether σ is a UVS for ψ' reduces to checking whether:

$$\phi(\vec{S}, \vec{R}) \wedge \gamma(\vec{P}) \ \equiv\ \phi^{\sigma}(\vec{S}, \vec{R}) \wedge \gamma^{\sigma}(\vec{P}). \tag{C.2}$$

It is immediate to observe that, if $\gamma(\vec{P})$ is a contradiction, formula (C.2) holds. Hence, what remains to be proven is that, if formula (C.2) holds, then $\gamma(\vec{P})$ must be a contradiction. Suppose this is not the case, i.e., there exists an extension $\overline{\vec{P}}$ for predicates in $\vec{P}$ for which $\gamma(\overline{\vec{P}})$ evaluates to true. Since, by hypothesis, formula (C.1) holds, there exists an extension $\langle \vec{\overline{S}}, \mathcal{I}\rangle$ for $\langle \vec{S}, \vec{R}\rangle$ such that $\phi(\vec{\overline{S}}, \mathcal{I})$ is true, but $\phi^{\sigma}(\vec{\overline{S}}, \mathcal{I})$ is false (or vice versa). This implies that the extension $\langle \vec{\overline{S}}, \mathcal{I}, \vec{\overline{P}}\rangle$ makes the left part of formula (C.2) true, and the right part false (or vice versa). In both cases, formula (C.2) does not clearly hold.

What it has been proven is that an arbitrary given first-order closed formula $\gamma(\vec{P})$ is a contradiction if and only if a fixed UVT σ is a UVS for the specification $\exists \vec{S}\ \phi(\vec{S}, \vec{R}) \wedge \gamma(\vec{P})$. Since the former problem is undecidable (9), the latter is undecidable as well. □

Proof of Proposition 4.4.1. First of all, we observe that for each input instance $\mathcal{I}$, if problem $\exists \vec{S}\ \phi(\vec{S}, \mathcal{I})$ has solutions, then, for every UVT σ and positive integer k, also $\exists \vec{S}\ (\phi(\vec{S}, \mathcal{I}))^{\sigma^k}$ has solutions, where σ^k denotes k consecutive applications of function σ. In fact, supposing $\vec{\overline{S}} = \{\overline{S_1}, \dots, \overline{S_n}\}$ a solution of the former, the extension $\sigma^k(\vec{\overline{S}}) = \{\sigma^k(\overline{S_1}), \dots, \sigma^k(\overline{S_n})\}$ is a solution of the latter.

Now, suppose that, for a given input instance $\mathcal{I}$, $\exists \vec{S}\ \phi(\vec{S}, \mathcal{I})$ has solutions; this implies that $\phi(\vec{S}, \mathcal{I})$ is satisfiable. From point 2 of Definition 4.4.1 it follows that also $\bigvee_{\vec{\sigma} \in \sigma^*} (\phi(\vec{S}, \mathcal{I}) \wedge \beta(\vec{S}))^{\vec{\sigma}}$ is satisfiable. Hence, there exists a $k \geq 0$ such that $(\phi(\vec{S}, \mathcal{I}) \wedge \beta(\vec{S}))^{\sigma^k}$ holds. Since both σ and σ^{-1} are UVTs for the problem $\exists \vec{S}\ \phi(\vec{S}, \mathcal{I}) \wedge \beta(\vec{S})$, from the observation above, it follows that *all* $(\phi(\vec{S}, \mathcal{I}) \wedge \beta(\vec{S}))^{\vec{\sigma}}$, with $\vec{\sigma} \in \sigma^*$ are satisfiable. In particular, for $\vec{\sigma} = \langle\rangle$, $\exists \vec{S}\ \phi(\vec{S}, \mathcal{I}) \wedge \beta(\vec{S})$, i.e., the rewritten problem, has solutions. □

Lemma C.0.1 (Alternate formulation of condition 1 of Definition 4.4.1). *Let $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$, be a specification, with $\vec{S} = \{S_1, \dots S_n\}$, S_i monadic for every $i \in [1, n]$, and input schema $\vec{R}$, let σ be a UVS for ψ, and $\beta(\vec{S})$ a closed (except for $\vec{S}$) formula. Formula $\beta(\vec{S})$ respects condition 1 of Definition 4.4.1 if and only if the following holds:*

$$\phi(\vec{S}, \vec{R}) \ \not\models\ \beta(\vec{S}) \leftrightarrow \beta(\vec{S})^{\sigma} \tag{C.3}$$

i.e., there exists a model $\langle \vec{\overline{S}}, \vec{\overline{R}}\rangle$ of ϕ that makes $\beta(\vec{\overline{S}})$ and $\beta(\vec{\overline{S}})^{\sigma}$ not equivalent.

Proof. If part. Since $\phi(\vec{S}, \vec{R}) \equiv \phi(\vec{S}, \vec{R})^{\sigma}$ (because σ is a UVS for ψ and because of Proposition 4.3.1), condition 1 of Definition 4.4.1 can be rewritten as:

$$\phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S}) \not\equiv \phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S})^{\sigma}.$$

If the above formula holds, then there exists an interpretation $\langle \vec{\overline{S}}, \vec{\overline{R}}\rangle$ that makes its left side $\phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S})$ true, and its right side $\phi(\vec{S}, \vec{R}) \wedge \beta(\vec{S})^{\sigma}$ false (or vice versa). Furthermore, this interpretation makes $\phi(\vec{S}, \vec{R})$ true, and $\beta(\vec{S})$ true and $\beta(\vec{S})^{\sigma}$ false (or vice versa). Hence, it proves that formula (C.3) holds.

Only if part. Suppose that formula (C.3) holds, and that $\langle \vec{\overline{S}}, \vec{\overline{R}}\rangle$ is a model of ϕ that makes $\beta(\vec{\overline{S}})$ true and $\beta(\vec{\overline{S}})^{\sigma}$ false (or vice versa). It is straightforward to check that such an interpretation makes the left side of condition 1 of Definition 4.4.1 true and the right side false (or vice versa). □

Lemma C.0.2 (Alternate formulation of condition 2 of Definition 4.4.1). *Given ψ, σ, and $\beta(\vec{S})$ as above, formula $\beta(\vec{S})$ respects condition 2 of Definition 4.4.1 if and only if the following holds:*

$$\phi(\vec{S},\vec{R}) \models \bigvee_{\vec{\sigma}\in\sigma^*} \beta(\vec{S})^{\vec{\sigma}}. \tag{C.4}$$

Proof. Since for every $\vec{\sigma} \in \sigma^*$, $\phi(\vec{S},\vec{R})^{\vec{\sigma}} \equiv \phi(\vec{S},\vec{R})$ (because σ is a UVS for ψ and because of Proposition 4.3.1), condition 2 of Definition 4.4.1 can be rewritten as:

$$\phi(\vec{S},\vec{R}) \models \phi(\vec{S},\vec{R}) \wedge \bigvee_{\vec{\sigma}\in\sigma^*} \beta(\vec{S})^{\vec{\sigma}},$$

which is clearly equivalent to (C.3). □

Corollary C.0.1 (Sufficient condition for condition 2 of Definition 4.4.1). *Given ψ, σ, and $\beta(\vec{S})$ as above, if $\bigvee_{\vec{\sigma}\in\sigma^*} \beta(\vec{S})^{\vec{\sigma}}$ is a tautology, then formula $\beta(\vec{S})$ respects condition 2 of Definition 4.4.1.*

Proof. Straightforward, from Lemma C.0.2. □

Example 4.5.1. We show how symmetry-breaking formulae presented in Example 4.5.1 for the Graph 3-coloring problem respect both conditions of Definition 4.4.1. ϕ denotes the first-order part of the Graph 3-coloring specification, cf. Example 1.2.2.

1. Formula $\beta^{G,B}_{least}(R,G,B) \doteq \forall Y\ B(Y) \rightarrow \exists X\ G(X) \wedge X < Y$, with respect to symmetry $\sigma^{G,B}$:

   Condition 1: We use Lemma C.0.1: to this end, $(\beta^{G,B}_{least}(R,G,B))^{\sigma^{G,B}}$ is

   $$\forall Y\ G(Y) \rightarrow \exists X\ B(X) \wedge X < Y.$$

   Consider an input graph for which $edge = \{(v,v)\}$, and the color assignment $\overline{R},\overline{G},\overline{B}$ such that $\overline{R} = \overline{B} = \emptyset, \overline{G} = \{v\}$. $\overline{R},\overline{G},\overline{B}$ is a good 3-coloring of the input graph, hence it satisfies $\phi(R,G,B,edge)$.

   Furthermore, interpretation $\langle\overline{R},\overline{G},\overline{B},edge\rangle$ makes $\beta^{G,B}_{least}(R,G,B)$ true and $(\beta^{G,B}_{least}(R,G,B))^{\sigma^{G,B}}$ false, implying that formula (C.3) holds. By Lemma C.0.1, $\beta^{G,B}_{least}(R,G,B)$ respects condition 1 of Definition 4.4.1.

   Condition 2: We use Lemma C.0.2: to this end, it is enough to observe that $\beta^{G,B}_{least}(R,G,B) \vee (\beta^{G,B}_{least}(R,G,B))^{\sigma^{G,B}}$ can be rewritten as:

   $$\begin{array}{l}\forall Y\ \neg B(Y) \ \vee\ (\exists X\ G(X) \wedge X < Y) \ \vee \\ \qquad \neg G(Y) \ \vee\ (\exists X\ B(X) \wedge X < Y)\end{array}$$

   and then as:

$$\begin{aligned}\forall Y\ (B(Y) \wedge G(Y)) \ \rightarrow\ & (\exists X\ G(X) \wedge X < Y)\ \vee \\ & (\exists X\ B(X) \wedge X < Y)\,.\end{aligned}$$

Since for every every model $\langle \overline{R}, \overline{G}, \overline{B}, edge \rangle$ of ϕ it holds that $\forall Y\ \neg\big(\overline{B}(Y) \wedge \overline{G}(Y)\big)$, cf. clause (1.9), then the above formula is satisfied by all such interpretations. Hence, formula (C.3) holds, and, by Lemma C.0.2, $\beta^{G,B}_{least}(R,G,B)$ respects condition 2 of Definition 4.4.1.

2. Formula $\beta^{G,B}_{\leq}(R,G,B)\ \doteq\ |G| \leq |B|$, with respect to symmetry $\sigma^{G,B}$:

   Condition 1: We use Lemma C.0.1: to this end, we first observe that $(\beta^{G,B}_{\leq}(R,G,B))^{\sigma^{G,B}}$ is $|B| \leq |G|$.

   Consider an input graph for which $edge = \{(v,v)\}$, and the color assignment $\overline{R}, \overline{G}, \overline{B}$ such that $\overline{R} = \overline{G} = \emptyset, \overline{B} = \{v\}$. $\overline{R}, \overline{G}, \overline{B}$ is a good 3-coloring of the input graph, hence it satisfies $\phi(R,G,B,edge)$.

   Furthermore, interpretation $\langle \overline{R}, \overline{G}, \overline{B}, edge \rangle$ makes $\beta^{G,B}_{\leq}(R,G,B)$ true and $(\beta^{G,B}_{\leq}(R,G,B))^{\sigma^{G,B}}$ false, implying that formula (C.3) holds. By Lemma C.0.1, $\beta^{G,B}_{\leq}(R,G,B)$ respects condition 1 of Definition 4.4.1.

   Condition 2: We use Corollary C.0.1: to this end, it is enough to observe that $\beta^{G,B}_{\leq}(R,G,B) \vee (\beta^{G,B}_{\leq}(R,G,B))^{\sigma^{G,B}}$ is simply:

   $$|G| \leq |B|\ \vee |B| \leq |G|,$$

   hence a tautology, because of the total ordering over positive integers.

3. Formula $\beta^{R}_{\leq}(R,G,B)\ \doteq\ |R| \leq |G| \wedge |R| \leq |B|$, with respect to both symmetries $\sigma^{R,G}$ and $\sigma^{R,B}$.

   Condition 1: We must use the definition of multiple symmetry-breaking formula. As for condition 1, we have to check the following two conditions:

   (a) $(\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S})) \not\equiv (\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S}))^{\sigma^{R,G}}$;

   (b) $(\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S})) \not\equiv (\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S}))^{\sigma^{R,B}}$.

   As for condition (a), it becomes:

   $$\Big(\phi(\vec{S},\vec{R}) \wedge |R| \leq |G| \wedge |R| \leq |B|\Big)\ \not\equiv\ \Big(\phi(\vec{S},\vec{R}) \wedge |G| \leq |R| \wedge |G| \leq |B|\Big)\,.$$

   Consider an input graph for which $edge = \{(v,v)\}$, and a color assignment $\overline{R}, \overline{G}, \overline{B}$ such that $\overline{R} = \overline{B} = \emptyset, \overline{G} = \{v\}$. $\overline{R}, \overline{G}, \overline{B}$ is a good 3-coloring of the input graph, and interpretation $\langle \overline{R}, \overline{G}, \overline{B}, edge \rangle$ makes the left part of the non-equivalence true, and the right part false.

   As for condition (b), a similar argument holds, with color assignment $\overline{R}, \overline{G}, \overline{B}$ such that $\overline{R} = \overline{G} = \emptyset, \overline{B} = \{v\}$.

Condition 2: Since $\beta_{\leq}^{G,B}(R,G,B)$ is a multiple symmetry-breaking formula, we must check condition (4.3). To this end, it suffices to show that:

$$\beta_{\leq}^{G,B}(R,G,B) \ \vee\ (\beta_{\leq}^{G,B}(R,G,B))^{\sigma^{R,G}} \ \vee\ (\beta_{\leq}^{G,B}(R,G,B))^{\sigma^{R,B}}$$

is a tautology. The above formula can be rewritten as:

$$\begin{array}{l}(|R| \leq |G| \ \wedge\ |R| \leq |B|) \ \vee \\ (|G| \leq |R| \ \wedge\ |G| \leq |B|) \ \vee \\ (|B| \leq |G| \ \wedge\ |B| \leq |R|)\,.\end{array}$$

To show that it is a tautology, we intuitively observe that the first disjunct is satisfied by color assignments for which red is used to color the less number of nodes. Similarly the second and third disjuncts are satisfied by assignments for which green and blue, respectively, are used to color the less number of nodes. It is straightforward to observe that every color assignment must be satisfied by at least one of them.

4. Formula $\beta_{\leq}(R,G,B) \ \doteq\ |R| \leq |G| \wedge |R| \leq |B| \wedge |G| \leq |B|$, with respect to the three symmetries $\sigma^{R,G}$, $\sigma^{R,B}$, and $\sigma^{G,B}$.

   Conditions 1 and 2 of the multiple symmetry-breaking formula definition can be checked by using very similar arguments of the previous point.

$\square$

Example 4.5.2. To show that formulae (4.9) and (4.10) respect both conditions of Definition 4.4.1, a technique very similar to that of the previous example can be used. $\square$

Example 4.5.3. We show the encoding β_{lex^2} of the lex^2 constraint for the Social golfer specification in Example 4.5.3, and prove that it is a multiple symmetry-breaking formula.

To this end, given the Social golfer specification, we unfold the ternary guessed predicate $PLAY(\cdot,\cdot,\cdot)$ into $|player| \times |week|$ monadic predicates $PLAY_{P,W}$, hence leading to the usual players/weeks matrix model of the problem.

Assuming, without loss of generality, that players are identified by integer constants in the range $[1, np]$, with $np = |player|$, and weeks by constants $[1, nw]$, with $nw = |week|$, the unfolded specification is as follows:

$$
\begin{aligned}
&\exists PLAY_{1,1},\ldots,PLAY_{1,nw},\ldots PLAY_{np,nw}\\
&\qquad\bigwedge_{p=1}^{np}\bigwedge_{w=1}^{nw} \forall G\ PLAY_{p,w}(G)\ \rightarrow\ group(G)\ \wedge\\
&\qquad\bigwedge_{p=1}^{np}\bigwedge_{w=1}^{nw} \exists G\ PLAY_{p,w}(G)\ \wedge\\
&\qquad\bigwedge_{p=1}^{np}\bigwedge_{w=1}^{nw} \forall G,G'\ PLAY_{p,w}(G)\wedge PLAY_{p,w}(G')\ \rightarrow\ G=G'\ \wedge\\
&\qquad\bigwedge_{p=1}^{np}\bigwedge_{w=1}^{nw}\bigwedge_{\substack{p'=1\\p'\neq p}}^{np}\bigwedge_{\substack{w'=1\\w'\neq w}}^{nw} \forall G,G'\ \left(PLAY_{p,w}(G)\wedge PLAY_{p',w}(G)\right)\rightarrow\\
&\qquad\qquad\neg\left[PLAY_{p,w'}(G')\wedge PLAY_{p',w'}(G')\right]\ \wedge\\
&\qquad\bigwedge_{w=1}^{nw}\bigwedge_{w'=1}^{nw} \forall G,G'\ group(G)\wedge group(G')\ \rightarrow\\
&\qquad\qquad|\{p\in[1,np]:PLAY_{p,w}(G)\}| = |\{p\in[1,np]:PLAY_{p,w'}(G')\}|.
\end{aligned}
$$

Let us consider, for any two fixed $\overline{p}$ and $\overline{\overline{p}}$ in $[1,np]$, $\overline{p}\neq\overline{\overline{p}}$, UVT $\sigma^{\overline{p},\overline{\overline{p}}}$, defined as follows:

$$
\sigma^{\overline{p},\overline{\overline{p}}}(PLAY_{p,w})\ =\ \begin{cases}\text{if } p=\overline{p}, \text{ then } PLAY_{\overline{\overline{p}},w}\\ \text{if } p=\overline{\overline{p}}, \text{ then } PLAY_{\overline{p},w}\\ \text{the identify function, otherwise}\end{cases}
$$

In other words, for each week $w\in[1,nw]$, $\sigma^{\overline{p},\overline{\overline{p}}}$ swaps predicates $PLAY_{\overline{p},w}$ and $PLAY_{\overline{\overline{p}},w}$, thus exchanging the whole schedulings for players $\overline{p}$ and $\overline{\overline{p}}$. Moreover, for every pair $\overline{p}$ and $\overline{\overline{p}}$ in $[1,np]$, $\sigma^{\overline{p},\overline{\overline{p}}}$ is a symmetry for the unfolded specification. This can be checked by replacing every occurrence of $PLAY_{\overline{p},w}$ with $PLAY_{\overline{\overline{p}},w}$ and vice versa (cf. Proposition 4.3.1), and by observing that every clause of the former specification occurs in the latter and vice versa (syntactic equivalence).

Once a linear order "<" over groups has been fixed, a symmetry-breaking formula for $\sigma^{\overline{p},\overline{\overline{p}}}$ is as follows:

$$
\begin{aligned}
\beta_{lex}^{\overline{p},\overline{\overline{p}}}\ \doteq\ &\bigvee_{\overline{w}=1}^{nw}\Bigg[\bigwedge_{\substack{w=1\\w<\overline{w}}}^{nw}\forall G,G'\ \left(PLAY_{\overline{p},w}(G)\wedge PLAY_{\overline{\overline{p}},w}(G')\right)\ \rightarrow\ G=G'\ \wedge\\
&\qquad\forall G,G'\ \left(PLAY_{\overline{p},\overline{w}}(G)\wedge PLAY_{\overline{\overline{p}},\overline{w}}(G')\right)\ \rightarrow\ G<G'\Bigg]\ \vee\\
&\bigwedge_{w=1}^{nw}\forall G,G'\ \left(PLAY_{\overline{p},w}(G)\wedge PLAY_{\overline{\overline{p}},w}(G')\right)\ \rightarrow\ G=G'
\end{aligned}
$$

and forces the lexicographic ordering on the whole schedulings for players $\overline{p}$ and $\overline{\overline{p}}$ (actually, this formulation is more general than needed for this problem, because, e.g., the other constraints do not allow for the schedulings to be equal). The version of this symmetry-breaking formula for the original specification can be derived by re-folding it, hence obtaining:

$$\Big[\exists \overline{W}\ \forall W\ \ W < \overline{W}\ \rightarrow\ \forall G, G'\ \Big(PLAY(\overline{P}, W, G) \wedge PLAY(\overline{\overline{P}}, W, G')\Big)\ \rightarrow\ G = G') \wedge$$
$$\forall G, G'\ \Big(PLAY(\overline{P}, \overline{W}, G) \wedge PLAY(\overline{\overline{P}}, \overline{W}, G')\Big)\ \rightarrow\ G < G'\Big]\ \vee$$
$$\forall W, G, G'\ \Big(PLAY(\overline{P}, W, G) \wedge PLAY(\overline{\overline{P}}, W, G')\Big)\ \rightarrow\ G = G'.$$

$\beta_{lex}^{\overline{p},\overline{\overline{p}}}$ deals with only one symmetry, i.e., $\sigma^{\overline{p},\overline{\overline{p}}}$. We now define $\beta_{lex}^{players}$ as follows:

$$\beta_{lex}^{players}\ \doteq\ \bigwedge_{\substack{\overline{p},\overline{\overline{p}}=1\\ \overline{p}<\overline{\overline{p}}}}^{np} \beta_{lex}^{\overline{p},\overline{\overline{p}}}, \tag{C.5}$$

and prove that $\beta_{lex}^{players}$ is a multiple symmetry-breaking formula, breaking all symmetries $\sigma^{\overline{p},\overline{\overline{p}}}$.

As for the first condition, we use must show that $\beta_{lex}^{players}$ breaks all those symmetries. We do this by using Lemma C.0.1 for any pair $\overline{p}$ and $\overline{\overline{p}}$ of players. To this end, we show an instance of the Social golfer problem and a solution that satisfies $\beta_{lex}^{players}$ but does not satisfy $(\beta_{lex}^{players})^{\sigma^{\overline{p},\overline{\overline{p}}}}$, for any pair $\overline{p}$ and $\overline{\overline{p}}$.

The instance has 4 players ($np = 4$), 3 weeks ($nw = 3$), and 2 groups, and a solution is as follows:

$$\begin{array}{lll}
PLAY_{1,1}(1), & PLAY_{1,2}(1) & PLAY_{1,3}(1)\\
PLAY_{2,1}(1), & PLAY_{2,2}(2) & PLAY_{2,3}(2)\\
PLAY_{3,1}(2), & PLAY_{3,2}(1) & PLAY_{3,3}(2)\\
PLAY_{4,1}(2), & PLAY_{4,2}(2) & PLAY_{4,3}(1)
\end{array}$$

i.e., player 1 plays in group 1 on weeks 1, 2, 3, player 2 plays in group 1 on week 1, and in group 2 on weeks 2 and 3, etc.

This solution satisfies $\beta_{lex}^{players}$ since, for any two players $\overline{p}$ and $\overline{\overline{p}}$, scheduling of player $\overline{p}$ is lexicographically less than that of player $\overline{\overline{p}}$ if and only if $\overline{p} < \overline{\overline{p}}$ (schedulings are the "rows" in the matrix above). On the other hand, $(\beta_{lex}^{players})^{\sigma^{\overline{p},\overline{\overline{p}}}}$ is obtained from $\beta_{lex}^{players}$ by uniformly exchanging every occurrence of $PLAY_{\overline{p},w}$ with $PLAY_{\overline{\overline{p}},w}$ and vice versa, for every w. Hence, $(\beta_{lex}^{players})^{\sigma^{\overline{p},\overline{\overline{p}}}}$ imposes a lexicographic ordering on players schedulings with the roles of $\overline{p}$ $\overline{\overline{p}}$ reversed (i.e., if $\overline{p}$ was less than $\overline{\overline{p}}$, now it is greater, and vice versa). Hence, the scheduling above cannot satisfy $(\beta_{lex}^{players})^{\sigma^{\overline{p},\overline{\overline{p}}}}$.

As for the second condition, we use the generalization of Lemma C.0.2 to the case of multiple symmetries, showing that, given $\Sigma = \{\sigma^{\overline{p},\overline{\overline{p}}},\ \overline{p} < \overline{\overline{p}},\ \overline{p},\overline{\overline{p}} \in [1, np]\}$, every solution for any instance of the Social golfer problem makes the following formula true:

$$\bigvee_{\vec{\sigma} \in \Sigma^*} (\beta_{lex}^{players})^{\vec{\sigma}}$$

To this end, consider an arbitrary instance of the problem and a solution. This is encoded in an extension for predicates $PLAY_{p,w}$, $p \in [1, np]$, $w \in [1, nw]$ containing exactly one tuple each (the group in which player p plays in week w). This interpretation satisfies $(\beta_{lex}^{players})^{\vec{\sigma}}$ for $\vec{\sigma}$ built as follows:

1. $\vec{\sigma} = \langle\rangle$ (i.e., the empty sequence);
2. If there exist a pair $\langle PLAY_{\overline{p},1}, \ldots, PLAY_{\overline{p},nw}\rangle \not\leq_{lex} \langle PLAY_{\overline{\overline{p}},1}, \ldots, PLAY_{\overline{\overline{p}},nw}\rangle$ with $\overline{p} < \overline{\overline{p}}$:
   (a) For every week w, swap tuples of predicates $PLAY_{\overline{p},w}$ and $PLAY_{\overline{\overline{p}},w}$;
   (b) Add $\sigma^{\overline{p},\overline{\overline{p}}}$ to $\vec{\sigma}$.
   (c) Go to 2.
3. Return $\vec{\sigma}$.

We do not give a formal proof of the correctness of this algorithm, but intuitively observe that the idea is to make the solution lexicographically ordered by swapping non-ordered schedulings. $\vec{\sigma}$ will contain the sequence of such swaps.

As an example, consider again the instance with 4 players, 3 weeks, and 2 groups, and the following solution:

$$\begin{array}{lll}
PLAY_{1,1}(2), & PLAY_{1,2}(1) & PLAY_{1,3}(2) \\
PLAY_{2,1}(2), & PLAY_{2,2}(2) & PLAY_{2,3}(1) \\
PLAY_{3,1}(1), & PLAY_{3,2}(2) & PLAY_{3,3}(2) \\
PLAY_{4,1}(1), & PLAY_{4,2}(1) & PLAY_{4,3}(1)
\end{array}$$

We now build $\vec{\sigma}$ such that the above interpretation satisfies $(\beta_{lex}^{players})^{\vec{\sigma}}$.

Start with $\vec{\sigma} = \langle\rangle$. Since the scheduling for player 2 (second row) is not lexicographically less than that for player 3 (third row), we swap tuples of $PLAY_{2,w}$ and $PLAY_{3,w}$ for every week w, hence obtaining:

$$\begin{array}{lll}
PLAY_{1,1}(2), & PLAY_{1,2}(1) & PLAY_{1,3}(2) \\
PLAY_{2,1}(1), & PLAY_{2,2}(2) & PLAY_{2,3}(2) \\
PLAY_{3,1}(2), & PLAY_{3,2}(2) & PLAY_{3,3}(1) \\
PLAY_{4,1}(1), & PLAY_{4,2}(1) & PLAY_{4,3}(1)
\end{array}$$

and add $\sigma^{2,3}$ to $\vec{\sigma}$. By iterating, we build $\vec{\sigma}$ as $\langle\sigma^{2,3}, \sigma^{1,2}, \sigma^{3,4}, \sigma^{2,3}, \sigma^{1,2}\rangle$.

We observe that the proposed algorithm for generating $\vec{\sigma}$ is very naive and does not return the shortest $\vec{\sigma}$. As an example, a shorter sequence of swaps that orders the rows of the matrix in the solution above is $\langle \sigma^{1,4}, \sigma^{2,3}, \sigma^{3,4} \rangle$.

Let us now consider such a sequence (for simplicity the shorter one), and the corresponding $(\beta_{lex}^{players})^{\langle \sigma^{1,4}, \sigma^{2,3}, \sigma^{3,4} \rangle}$. This formula forces the following ordering on players schedulings:

$$\begin{aligned}
\langle PLAY_{4,1}, \ldots, PLAY_{4,nw} \rangle &\leq_{lex} \\
\langle PLAY_{3,1}, \ldots, PLAY_{3,nw} \rangle &\leq_{lex} \\
\langle PLAY_{1,1}, \ldots, PLAY_{1,nw} \rangle &\leq_{lex} \\
\langle PLAY_{2,1}, \ldots, PLAY_{2,nw} \rangle &.
\end{aligned}$$

Hence, the initial solution satisfies it.

The definition of $\beta_{lex}^{players}$, cf. formula (C.5), defines $np \times (np-1)/2$ lexicographic ordering constraints among different players schedulings. However, it is easy to prove that it is equivalent to the following one:

$$\bigwedge_{\overline{p}=1}^{np-1} \beta_{lex}^{\overline{p},\overline{p}+1}$$

that forces only a linear number of such constraints.

Analogously to $\sigma^{\overline{p},\overline{\overline{p}}}$, we can define transformations $\sigma^{\overline{w},\overline{\overline{w}}}$, that, for each player $p \in [1, np]$, swap predicates $PLAY_{p,\overline{w}}$ and $PLAY_{p,\overline{\overline{w}}}$, thus exchanging, for each player, the schedulings for weeks $\overline{w}$ and $\overline{\overline{w}}$. By using the same technique presented for $\sigma^{\overline{p},\overline{\overline{p}}}$, it is possible to show that all $\sigma^{\overline{w},\overline{\overline{w}}}$ are symmetries for the unfolded Social golfer specification. A multiple symmetry-breaking formula β_{lex}^{weeks} can be defined similarly to $\beta_{lex}^{players}$.

From $\beta_{lex}^{players}$ and β_{lex}^{weeks}, we finally define β_{lex^2} as follows:

$$\beta_{lex^2} \doteq \beta_{lex}^{players} \wedge \beta_{lex}^{weeks} \tag{C.6}$$

and prove that β_{lex^2} breaks all symmetries $\sigma^{\overline{p},\overline{\overline{p}}}$ and $\sigma^{\overline{w},\overline{\overline{w}}}$, with $\overline{p}$ and $\overline{\overline{p}}$ in $[1, np]$ and $\overline{w}$ and $\overline{\overline{w}}$ in $[1, nw]$.

As for condition 1, it suffices to show that there exists an instance and solution for the Social golfer problem that satisfies β_{lex^2}, but does not satisfy $(\beta_{lex^2})^{\sigma}$, for every $\sigma \in \{\sigma^{\overline{p},\overline{\overline{p}}},\ \overline{p}, \overline{\overline{p}} \in [1, np]\} \cup \{\sigma^{\overline{w},\overline{\overline{w}}},\ \overline{w}, \overline{\overline{w}} \in [1, nw]\}$.

An example is again the instance with 4 players, 3 weeks, and 2 groups, and the following solution:

$$\begin{array}{lll}
\mathit{PLAY}_{1,1}(1), & \mathit{PLAY}_{1,2}(1) & \mathit{PLAY}_{1,3}(1) \\
\mathit{PLAY}_{2,1}(1), & \mathit{PLAY}_{2,2}(2) & \mathit{PLAY}_{2,3}(2) \\
\mathit{PLAY}_{3,1}(2), & \mathit{PLAY}_{3,2}(1) & \mathit{PLAY}_{3,3}(2) \\
\mathit{PLAY}_{4,1}(2), & \mathit{PLAY}_{4,2}(2) & \mathit{PLAY}_{4,3}(1)
\end{array}$$

This solution clearly satisfies both $\beta_{lex}^{players}$ and β_{lex}^{weeks}, and so β_{lex^2}. However, for every $\sigma \in \{\sigma^{\overline{p},\overline{\overline{p}}},\ \overline{p},\overline{\overline{p}} \in [1,4]\} \cup \{\sigma^{\overline{w},\overline{\overline{w}}},\ \overline{w},\overline{\overline{w}} \in [1,3]\}$, this solution does not satisfy $(\beta_{lex^2})^{\sigma}$.

As for condition 2, instead, a simple generalization of the technique used for $\beta_{lex}^{players}$ and β_{lex}^{weeks} suffices, since every symmetric assignment can be obtained by a finite sequence of rows and columns swaps.

As a final remark, we observe that the proof presented here does not really rely on the specification for Social golfer. What is really important is that the unfolding of the specification leads to a row/column matrix of monadic guessed predicates, each of them forced to have exactly one tuple, and that all symmetries on rows and columns hold. Hence, the β_{lex^2} formula can be used also in other specifications, as that for the BIBD problem (cf. Example 4.5.4). $\square$

Appendix D

Chapter 5: proofs of results

Proof of Theorem 5.2.1.
If part. We show that if formula (5.1) is valid, then $\vec{P}$ functionally depends on $\vec{S}$. Actually, we prove that, if $\vec{P}$ does not functionally depend on $\vec{S}$, then formula (5.1) is not valid, i.e., there is an extension for $\vec{S}, \vec{P}, \vec{S'}, \vec{P'}, \vec{R}$ that falsifies it. Let us assume that $\vec{P}$ does not functionally depend on $\vec{S}$: this means, according to Definition 5.2.1, that there exists an instance $\mathcal{I}$ of $\vec{R}$ and two interpretations M and N of predicates in $(\vec{S}, \vec{P})$ such that $M \neq N$, $\mathcal{I}, M \models \phi$, $\mathcal{I}, N \models \phi$ but $M_{|\vec{S}} = N_{|\vec{S}}$, i.e., M and N are models of ϕ that differ only on the extension of predicates in $\vec{P}$. The interpretation $(M, N, \mathcal{I})$ (where M and N are the interpretations of predicates in $(\vec{S}, \vec{P})$ and in $(\vec{S'}, \vec{P'})$, respectively) makes the left side of implication (5.1) true, and the right side false. Thus this interpretation is not a model of formula (5.1).

Only if part. Here we show that if $\vec{P}$ functionally depends on $\vec{S}$, then formula (5.1) is valid. Actually, we prove that, if formula (5.1) is not valid, i.e., there is an extension for $\vec{S}, \vec{S'}, \vec{P}, \vec{P'}, \vec{R}$ that falsifies it, then $\vec{P}$ does not functionally depend on $\vec{S}$. Let us assume that an interpretation $(M, N, \mathcal{I})$ of predicates $\vec{S}, \vec{P}, \vec{S'}, \vec{P'}, \vec{R}$ exists that falsifies formula (5.1). This means that such an interpretation makes the left side of implication (5.1) true and the right side false. Thus, it is such that:

1. $M, \mathcal{I} \models \phi(\vec{S}, \vec{P}, \vec{R})$;
2. $N, \mathcal{I} \models \phi(\vec{S'}, \vec{P'}, \vec{R})$;
3. $M_{|\vec{P}} \neq N_{|\vec{P}}$;
4. $M_{|\vec{S}} = N_{|\vec{S}}$.

From points 1-4 and Definition 5.2.1, it follows that $\vec{P}$ does not functionally depend on $\vec{S}$, since M and N are two models of ϕ that differ only for the extension of predicates in $\vec{P}$. $\square$

Proof of Theorem 5.2.2.
We prove the statement by reducing it to the problem of checking whether an arbitrary closed first-order formula is a contradiction.

Let $\psi \doteq \exists \vec{S}\vec{P}\ \phi(\vec{S},\vec{P},\vec{R})$ be any fixed problem specification with input schema $\vec{R}$, such that $\vec{P}$ is functionally dependent on $\vec{S}$, so, by Theorem 5.2.1:

$$[\phi(\vec{S},\vec{P},\vec{R}) \land \phi(\vec{S'},\vec{P'},\vec{R}) \land \neg(\vec{S}\vec{P} \equiv \vec{S'}\vec{P'})] \rightarrow \neg(\vec{S} \equiv \vec{S'})$$

is a valid formula. Let $\gamma(\vec{R})$ be an arbitrary closed first-order formula on the relational vocabulary $\vec{R}$.

Consider the new specification $\psi' \doteq \exists \vec{S}\vec{P}\ \phi'(\vec{S},\vec{P},\vec{R})$, where $\phi'(\vec{S},\vec{P},\vec{R})$ is defined as $\phi(\vec{S},\vec{P},\vec{R}) \lor \gamma(\vec{R})$. From Theorem 5.2.1, $\vec{P}$ functionally depends on $\vec{S}$ with respect to specification ψ', if and only if

$$[\phi'(\vec{S},\vec{P},\vec{R}) \land \phi'(\vec{S'},\vec{P'},\vec{R}) \land \neg(\vec{S}\vec{P} \equiv \vec{S'}\vec{P})] \rightarrow \neg(\vec{S} \equiv \vec{S'})$$

is a valid formula, or, equivalently,

$$[\Big(\phi(\vec{S},\vec{P},\vec{R}) \lor \gamma(\vec{R})\Big) \land \Big(\phi(\vec{S'},\vec{P'},\vec{R}) \lor \gamma(\vec{R})\Big) \land \neg(\vec{S}\vec{P} \equiv \vec{S'}\vec{P'})] \rightarrow \neg(\vec{S} \equiv \vec{S'}) \quad \text{(D.1)}$$

is a valid formula. Since, by hypothesis, $\vec{P}$ functionally depends on $\vec{S}$ with respect to specification ψ, it follows that, if $\gamma(\vec{R})$ is a contradiction, then $\vec{P}$ functionally depends on $\vec{S}$ with respect to ψ'.

On the other hand, let us assume that an interpretation $\mathcal{I}$ for $\vec{R}$ exists such that $\gamma(\mathcal{I})$ is true. Consider a pair of interpretations M and N of $(\vec{S},\vec{P})$, such that $M \neq N$ but $M_{|\vec{S}} = N_{|\vec{S}}$. Thus, interpretation $(M, N, \mathcal{I})$ of predicates $(\vec{S},\vec{P})$, $(\vec{S'},\vec{P'})$, and $\vec{R}$ is not a model of formula (D.1). Since, for every interpretation $\mathcal{I}$ for $\vec{R}$ (with non-empty universe) a pair of interpretations M and N of the above kind can always be built, formula (D.1) is valid, i.e., $\vec{P}$ functionally depends on $\vec{S}$ with respect to ψ' if and only if $\gamma(\vec{R})$ is a contradiction. Since the latter problem is not decidable, cf., e.g., (9), the former is not decidable as well. $\square$

Appendix E

Chapter 6

E.1 First-order conditions for symmetry-breaking formulae in ESO

In Section 6.2.2 we have shown that, in order to check whether a given formula $\beta(\vec{S})$ breaks a symmetry σ for problem specification $\psi \doteq \exists \vec{S}\ \phi(\vec{S}, \vec{R})$, first-order theorem provers may in general not suffice. This happens, in particular, when formula $\beta(\vec{S})$ is second-order, but not first-order definable. In fact, in this case, conditions of Definition 4.4.1 become second-order (non-)equivalences. In the same section, we have presented an example of such a $\beta(\vec{S})$, namely $\beta_{\leq}^{G,B}(R, G, B)$ for the Graph 3-coloring specification and symmetry $\sigma^{G,B}$.

However, we have also argued that, in some circumstances, it is possible to define a sequence of first-order steps that allow to infer second-order (non-)equivalences of Definition 4.4.1.

In this section, we focus on candidate symmetry-breaking formulae $\beta(\vec{S})$ expressed in ESO, and show an example of first-order definable steps that imply that condition (4.1), i.e., point 1 of Definition 4.4.1 holds.

We assume that $\beta(\vec{S})$ is in the following form:

$$\beta(\vec{S}) \ \doteq\ \exists \vec{T}\ \gamma(\vec{S}, \vec{T}), \tag{E.1}$$

where $\gamma(\vec{S}, \vec{T})$ is a (possibly quantified) first-order formula open only with respect to the set of guessed predicates $\vec{S}$ of the whole specification, plus the set of existentially quantified predicates $\vec{T}$ in $\beta(\vec{S})$.

$\beta_{\leq}^{G,B}(R, G, B)$ belongs to this case, since, as discussed in Appendix A, it can be written in ESO as follows:

$$\begin{aligned}
\beta_{\leq}^{G,B}(R, G, B) \ \doteq\ \exists T\ & \forall X, Y\ T(X, Y) \ \rightarrow\ G(X) \wedge B(Y) \ \wedge && \text{(E.2)}\\
& \forall X \exists Y\ G(X) \ \rightarrow\ T(X, Y) \ \wedge && \text{(E.3)}\\
& \forall X, Y, Y'\ T(X, Y) \wedge T(X, Y') \ \rightarrow\ Y = Y' \ \wedge && \text{(E.4)}\\
& \forall X, X', Y\ T(X, Y) \wedge T(X', Y) \ \rightarrow\ X = X'. && \text{(E.5)}
\end{aligned}$$

The above formula evaluates to true if a total (E.3), injective (E.5) function (E.4) exists from the set of green nodes to that of blue ones (E.2), i.e., if and only if the number of green nodes is less than or equal to the number of blue ones in the assignment given by R, G, B.

For a formula in the form (E.1), since $\vec{T}$ does occurs neither in $\phi(\vec{S},\vec{R})$ nor in $\phi(\vec{S},\vec{R})^\sigma$, condition (4.1) of Definition 4.4.1 becomes as follows:

$$\exists\vec{T}\ \left(\phi(\vec{S},\vec{R}) \wedge \gamma(\vec{S},\vec{T})\right) \not\equiv \exists\vec{T}\ \left(\phi(\vec{S},\vec{R}) \wedge \gamma(\vec{S},\vec{T})\right)^\sigma. \tag{E.6}$$

We immediately note that testing whether $\left(\phi \wedge \gamma(\vec{S},\vec{T})\right) \not\equiv \left(\phi \wedge \gamma(\vec{S},\vec{T})\right)^\sigma$ does not suffice. However, we can proceed as follows: our goal is to find an extension $\langle\vec{\vec{S}},\vec{\vec{R}}\rangle$ for predicates in $\vec{S}$ and $\vec{R}$ that is a model of the left side of (E.6) and not of the right side (or vice versa). Thus, we are interested in an interpretation $\langle\vec{\vec{S}},\vec{\vec{R}}\rangle$ for which an extension for predicates in $\vec{T}$ that satisfies $\phi(\vec{S},\vec{R}) \wedge \gamma(\vec{S},\vec{T})$ does exist, but an extension for predicates in $\vec{T}$ that satisfies $\left(\phi(\vec{S},\vec{R}) \wedge \gamma(\vec{S},\vec{T})\right)^\sigma$ does not (or vice versa).

To this end, we can use a first-order finite model finder to get an extension $\langle\vec{\vec{S}},\vec{\vec{R}},\vec{\vec{T}}\rangle$ for $\vec{S}$, $\vec{R}$ and $\vec{T}$ such that:

$$\langle\vec{\vec{S}},\vec{\vec{R}},\vec{\vec{T}}\rangle \models \phi(\vec{S},\vec{R}) \wedge \gamma(\vec{S},\vec{T}).$$

$\langle\vec{\vec{S}},\vec{\vec{R}}\rangle$ consists of an instance for the problem $\psi \doteq \exists\vec{S}\ \phi(\vec{S},\vec{R})$ plus a solution for that instance. Moreover, this solution satisfies the symmetry-breaking formula $\beta(\vec{S}) \doteq \exists\vec{T}\ \gamma(\vec{S},\vec{T})$.

Of course we are not yet guaranteed that for $\langle\vec{\vec{S}},\vec{\vec{R}}\rangle$ an extension $\vec{\vec{\vec{T}}}$ such that $\left(\phi(\vec{\vec{S}},\vec{\vec{R}}) \wedge \gamma(\vec{\vec{S}},\vec{\vec{\vec{T}}})\right)^\sigma$ evaluates to true does not exist. But, if $\gamma(\vec{\vec{S}},\vec{T})^\sigma$ is a contradiction, this happens, and so we have found an instance $\vec{\vec{R}}$ and an extension $\vec{\vec{S}}$ for $\vec{S}$ that is a solution for $\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S})$, but not for $\left(\phi(\vec{S},\vec{R}) \wedge \beta(\vec{S})\right)^\sigma$. This implies that condition (4.1) holds.

It is worthwhile noting that, in general, not every model found in the first step satisfies the additional condition that $\gamma(\vec{\vec{S}},\vec{T})^\sigma$ is a contradiction, so we may need to *backtrack* in order to look for another model. To reduce the number of such backtracks, a simple heuristic is to look, in the first step, for a model of $\phi(\vec{S},\vec{R}) \wedge \gamma(\vec{S},\vec{T}) \wedge \neg\gamma(\vec{S},\vec{T})^\sigma$. Summing up, the suggested procedure is the following:

1. Find $\langle\vec{\vec{S}},\vec{\vec{R}},\vec{\vec{T}}\rangle$ such that:

$$\langle\vec{\vec{S}},\vec{\vec{R}},\vec{\vec{T}}\rangle \models \phi(\vec{S},\vec{R}) \wedge \gamma(\vec{S},\vec{T}) \wedge \neg\gamma(\vec{S},\vec{T})^\sigma;$$

2. If $\gamma(\vec{\vec{S}},\vec{T})^\sigma \models \bot$, then condition (4.1) of Definition 4.4.1 holds; otherwise return to Step 1.

We observe that a *hybrid* strategy, that uses both a finite model finder and a theorem prover at the same time, can be effective in practice to perform Step 2.

As an example, we show how the procedure above works when checking whether $\beta_{\leq}^{G,B}(R,G,B)$ is symmetry-breaking for $\sigma^{G,B}$ in the Graph 3-coloring problem (cf. Example 1.2.2 for the problem specification).

We first look for an interpretation $\langle \overline{R}, \overline{G}, \overline{B}, \overline{edge}, \overline{T}\rangle$ such that $\overline{R}, \overline{G}, \overline{B}$ define a good coloring of the graph encoded by relation $\overline{edge}$, and $\overline{T}$ is a total and injective function from tuples in $\overline{R}$ to those in $\overline{G}$, but not vice-versa, cf. Step 1 of the procedure above. One of the smallest examples of such an extension is the following:

$$\overline{R} = \{\}; \qquad \overline{G} = \{v\}; \qquad \overline{B} = \{\} \qquad \overline{edge} = \{\}.$$

Now, consider formula $\gamma(\overline{R}, \overline{G}, \overline{B}, T)^{\sigma^{R,G}}$, with $\overline{R}, \overline{G}, \overline{B}$ as above. This formula is satisfiable if and only if a total and injective function T from tuples in $\overline{G}$ to those in $\overline{R}$ exists, that is, if and only if $|\overline{R}| \geq |\overline{G}|$. But of course, by construction, this is false (in Step 1 we synthesized $\langle \overline{R}, \overline{G}, \overline{B}\rangle$ such that $|\overline{R}| < |\overline{G}|$).

Hence, we have found an interpretation for predicates in $\vec{S}$ and in $\vec{R}$, namely $\langle \overline{R}, \overline{G}, \overline{B}, \overline{edge}\rangle$ that is a model of the left part of condition (E.6) and not of its right part. Thus, the second-order non-equivalence stated in (E.6) holds.

E.2 OTTER code for the examples

E.2.1 Detecting symmetries

Graph 3-coloring problem

```
set(auto).
formula_list(usable).

% Definition of phi
covering <-> (all x (R(x) | G(x) | B(x))).
disjRG   <-> (all x (R(x) -> - G(x))).
disjRB   <-> (all x (R(x) -> - B(x))).
disjBG   <-> (all x (B(x) -> - G(x))).
goodColR <-> (all x y (x!= y & R(x) & R(y) -> - Edge(x,y))).
goodColG <-> (all x y (x!= y & G(x) & G(y) -> - Edge(x,y))).
goodColB <-> (all x y (x!= y & B(x) & B(y) -> - Edge(x,y))).

3col   <-> (covering & disjRG & disjRB & disjBG & goodColR & goodColG & goodColB).

% Definition of phi^(sigma)
covering_sigma <-> (all x (G(x) | R(x) | B(x))).
disjGR_sigma   <-> (all x (G(x) -> - R(x))).
disjGB_sigma   <-> (all x (G(x) -> - B(x))).
disjBR_sigma   <-> (all x (B(x) -> - R(x))).
goodColG_sigma <-> (all x y (x!= y & G(x) & G(y) -> - Edge(x,y))).
goodColR_sigma <-> (all x y (x!= y & R(x) & R(y) -> - Edge(x,y))).
goodColB_sigma <-> (all x y (x!= y & B(x) & B(y) -> - Edge(x,y))).
```

```
3col_sigma   <-> (covering_sigma & disjGR_sigma & disjGB_sigma & disjBR_sigma &
                  goodColG_sigma & goodColR_sigma & goodColB_sigma).

% Conjecture: Is 3col equivalent to 3col_sigma?
-(3col <-> 3col_sigma).

end_of_list.
```

E.2.2 Detecting functional dependencies

Sailco inventory problem

```
set(auto).
formula_list(usable).

% Definition of the Sailco specification
  initial <-> inv(0) = inventory.
  regular <-> (all t (regulBoat(t) <= capacity)).
  allBoat <-> (all t (t > 0 -> regulBoat(t) + extraBoat(t) - demand(t) =
                     inv(t) - inv(t-1))).
  sailco  <-> allBoat & initial & regular.

% Definition of specification_prime
  initial_prime <-> inv_prime(0) = inventory.
  regular_prime <-> (all t (regulBoat_prime(t) <= capacity)).
  allBoat_prime <-> (all t (t > 0 -> regulBoat_prime(t) + extraBoat_prime(t) - demand(t) =
                           inv_prime(t) - inv_prime(t-1))).
  sailco_prime  <-> allBoat_prime & initial_prime & regular_prime.

% Induction principle for inv, inv_prime
  equalDiscrete <-> (inv(0) = inv_prime(0) &
                    (all t (t > 0 -> (inv(t) - inv(t-1)) =
                           (inv_prime(t) - inv_prime(t-1))))).
  induction <-> (equalDiscrete -> (all t (inv(t) = inv_prime(t)))).

% Extensions are equivalent on the "dependent" functions
  equivDep <-> (all t (inv(t) = inv_prime(t))).

% Extensions are equivalent on the "defining" functions
  equivDef <-> (all x (regulBoat(x) = regulBoat_prime(x) &
                       extraBoat(x) = extraBoat_prime(x))).

% Extensions are equivalent on all guessed functions
  equivAll <-> equivDep & equivDef.

% Conjecture: Is inv() dependent on regulBoat() and extraBoat()?
  -( (sailco & sailco_prime & induction & -equivAll) -> -equivDef ).
end_of_list.
```

HP 2D-Protein folding problem (model with new guessed predicate for non crossing)

```
set(auto).
formula_list(usable).


% Definition of the HP 2D-Protein folding specification

% Initial position of the string
  initpos <->  (X(0) = 0 & Y(0) = 0).

% Channelling constraints for X() and Y()
  channXY <-> (all t ( (t > 0) -> (
       (Moves(t-1) = N -> (X(t) = X(t-1) & Y(t)=Y(t-1)+1)) &
       (Moves(t-1) = S -> (X(t) = X(t-1) & Y(t)=Y(t-1)-1)) &
       (Moves(t-1) = E -> (X(t)=X(t-1)+1 & Y(t) = Y(t-1))) &
       (Moves(t-1) = W -> (X(t)=X(t-1)-1 & Y(t) = Y(t-1)))
  ))).
domMoves <-> (all t (Moves(t) = N  |  Moves(t) = S | Moves(t) = E | Moves(t) = W)).

% No overlap for for X() and Y()
nooverlap <-> (all t1 t2 (
                (t1 != t2) -> ( (X(t1) != X(t2)) | (Y(t1) != Y(t2)) )
      )).

protein <-> domMoves & initpos & channXY & nooverlap.


% Definition of specification_prime

% Initial position of the string
  initpos_prime <-> (X_prime(0) = 0 & Y_prime(0) = 0).

% Channelling constraints for X_prime() and Y_prime()
  channXY_prime <-> (all t ( (t > 0) -> (
        (Moves_prime(t-1) = N -> (X_prime(t) = X_prime(t-1) & Y_prime(t)=Y_prime(t-1)+1)  ) &
        (Moves_prime(t-1) = S -> (X_prime(t) = X_prime(t-1) & Y_prime(t)=Y_prime(t-1)-1)  ) &
        (Moves_prime(t-1) = E -> (X_prime(t)=X_prime(t-1)+1 & Y_prime(t) = Y_prime(t-1))  ) &
        (Moves_prime(t-1) = W -> (X_prime(t)=X_prime(t-1)-1 & Y_prime(t) = Y_prime(t-1))  )
  ))).
domMoves_prime <-> (all t (Moves_prime(t) = N  |  Moves_prime(t) = S |
                           Moves_prime(t) = E | Moves_prime(t) = W)).

% No overlap for for X_prime() and Y_prime()
nooverlap_prime <-> (all t1 t2 (
                       (t1 != t2) -> ( (X_prime(t1) != X_prime(t2)) |
                       (Y_prime(t1) != Y_prime(t2)) )
            )).

protein_prime <-> domMoves_prime & initpos_prime &
                  channXY_prime & nooverlap_prime.
```

```
%indentity <-> (
% (all t (((t+1)-1) = t )) &
% (all t ( ((t-1)+1) = t )) ).

identityX <-> ( (all t ((X(t)+1)-1 = X(t))) & (all t ((X(t)-1)+1 = X(t))) &
                  (all t ((X_prime(t)+1)-1 = X_prime(t))) &
                  (all t ((X_prime(t)-1)+1 = X_prime(t)))
).
identityY <-> ( (all t ((Y(t)+1)-1 = Y(t))) & (all t ((Y(t)-1)+1 = Y(t))) &
                  (all t ((Y_prime(t)+1)-1 = Y_prime(t))) &
                  (all t ((Y_prime(t)-1)+1 = Y_prime(t)))
).
equalDiscreteX <-> X(0) = X_prime(0) & (all t ( t > 0 ->
                      ( ( (X(t) = X(t-1)+1) & (X_prime(t) = X_prime(t-1)+1) &
                           X(t-1) = X_prime(t-1) )  |
                        ( (X(t) = X(t-1)-1) & (X_prime(t) = X_prime(t-1)-1) &
                           X(t-1) = X_prime(t-1) ) )
)).
inductionX <-> (equalDiscreteX -> (all t ( X(t) = X_prime(t) ))).

equalDiscreteY <-> Y(0) = Y_prime(0) & (all t ( t > 0 ->
                      ( ( (Y(t) = Y(t-1)+1) & (Y_prime(t) = Y_prime(t-1)+1) &
                           Y(t-1) = Y_prime(t-1) )  |
                        ( (Y(t) = Y(t-1)-1) & (Y_prime(t) = Y_prime(t-1)-1) &
                           Y(t-1) = Y_prime(t-1) ) )
)).
inductionY <-> (equalDiscreteY -> (all t (Y(t) = Y_prime(t)))).


movesDiff <-> (N != E & N != S  &  N != W  &  E != S  & E != W  &  S != W).

axioms <-> identityX & identityY    %% identity
              & inductionX & inductionY & movesDiff.

% Extensions are equivalent on the "dependent" functions
  equivDep <-> ((all t (X(t) = X_prime(t))) & (all t (Y(t) = Y_prime(t)))).

% Extensions are equivalent on the "defining" functions
  equivDef <-> (all t (Moves(t) = Moves_prime(t))).

% Extensions are equivalent on all guessed functions
  equivAll <-> equivDep & equivDef.

% Conjecture: are X() and Y() dependent on Moves()?
-( (protein & protein_prime & axioms & -equivAll) -> -equivDef ).
end_of_list.
```

Appendix F

Chapter 7: proofs of results

Proof of Lemma 7.3.1. Since variables in $\vec{X}$ do not occur in the sub-formula α, and since variables in α can be safely renamed to avoid conflicts with variables in $\vec{X}$, formula (7.7) can be rewritten as:

$$\forall \vec{M}\ \forall \vec{X}\ \Big(\beta(\vec{X})\ \vee\ \alpha(\vec{M})\ \vee\ \gamma(\vec{X}, \vec{M})\Big)\,.$$

Furthermore, $\forall \vec{X}$ can be switched with $\forall \vec{M}$, and since no predicate in $\vec{M}$ occurs in β, the latter quantifier can be pushed in, thus obtaining:

$$\forall \vec{X}\ \beta(\vec{X})\ \vee\ \forall \vec{M}\ \Big(\alpha(\vec{M})\ \vee\ \gamma(\vec{X}, \vec{M})\Big)\,.$$

By using the definition of $view(\vec{X})$ as in (7.9), the thesis follows. □

Appendix G

Chapter 8: *NP-Alg* and ConSql

G.1 *NP-Alg* expressions

In this section we collect and give the formal definitions of useful *NP-Alg* expressions, some of them used in Section 8.3, in order to let *NP-Alg* deal with functions, integers, and orderings.

- $\mathit{empty}^{(1)}(R) \quad \doteq$

$$DOM - \pi_{\$1}(DOM \times R^{(k)},) \tag{G.1}$$

  evaluates to the empty relation if R is a non-empty one (and vice versa).

- $\mathit{complement}^{(k)}(R^{(k)}) \quad \doteq$

$$\rho_{\substack{\$1 \rightarrow R.\$1 \\ \vdots \\ \$k \rightarrow R.\$k}} (DOM^k - R), \tag{G.2}$$

  evaluates to the active complement (with respect to DOM^k) of R (ρ is the field-renaming operator).

- $\mathit{failPartition}^{(1)}(N^{(k)}, P_1^{(k)}, \ldots, P_n^{(k)})$

  evaluates to the empty relation if and only if $\{P_1^{(k)}, \ldots, P_n^{(k)}\}$ is a partition of N (cf. Example 8.3.1 for its definition).

- $\mathit{failSuccessor}^{(1)}(SUCC^{(2k)}, N^{(k)})$

  evaluates to the empty relation if and only if $SUCC$ encodes a correct successor relation on elements in N, i.e., a 1-1 correspondence with the interval $[1, |N|]$ (essentially by checking whether $SUCC$ is an Hamiltonian path on the graph with edges defined by $N \times N$) (cf. Example 3.2.1 for an ESO specification of the HP problem).

- $\mathit{failPermutation}^{(1)}(\mathit{PERM}^{(2k)}, N^{(k)})$

  evaluates to the empty relation if and only if *PERM* is a permutation of the elements in $N^{(k)}$: the ordering sequence is given by the first k columns of *PERM*.

- $\mathit{failFunction}(\mathit{FUN}^{(2)}, D^{(1)}, R^{(1)}) \quad \doteq$

$$\left(\pi_{\$1}(\mathit{FUN}) - D\right) \cup \left(\pi_{\$2}(\mathit{FUN}) - R\right) \cup \pi_{\$1}\left(\mathit{FUN} \underset{\substack{\$1=\$1\\ \wedge\\ \$2\neq\$2}}{\bowtie} \mathit{FUN}\right) \tag{G.3}$$

  evaluates to the empty relation if and only if *FUN* is a function, i.e., a monodrome relation from D to R. The first and the second subexpressions check whether tuples in *FUN* are in the cartesian product $D \times R$, and the third checks whether *FUN* is monodrome (for the sake of simplicity, this and the following definitions are for $d = r = 1$, but their extensions to arbitrary d and r are straightforward).

- $\mathit{failTotal}(\mathit{FUN}^{(2)}, D^{(1)}, R^{(1)}) \quad \doteq$

$$D \; - \; \pi_{\$1}(\mathit{FUN}) \tag{G.4}$$

  evaluates to the empty relation if and only if *FUN* is defined for every tuple in D.

- $\mathit{failSurjective}(\mathit{FUN}^{(2)}, D^{(1)}, R^{(1)}) \quad \doteq$

$$R \; - \; \pi_{\$2}(\mathit{FUN}) \tag{G.5}$$

  evaluates to the empty relation if and only if the image of *FUN* is the whole relation R.

- $\mathit{failInjective}(\mathit{FUN}^{(2)}, D^{(1)}, R^{(1)}) \quad \doteq$

$$\mathit{FUN} \underset{\substack{\$1\neq\$1\\ \wedge\\ \$2=\$2}}{\bowtie} \mathit{FUN} \tag{G.6}$$

  evaluates to the empty relation if and only if *FUN* is such that different tuples in D are mapped into different tuples of R.

We remark that, since elements in R can be ordered (cf. expressions $\mathit{failSuccessor}()$ above), *FUN* is also an *integer function* from elements of D to the interval $[1, |R|]$. Integer functions are very useful for the specification of *resource allocation* problems, such as *Integer knapsack* (see also examples in Section 8.5.2).

- $\mathit{failGeqSize}(AUX, D, R) \quad \doteq$

$$\mathit{failFunction}(AUX, D, R) \cup \mathit{failSurjective}(AUX, D, R) \tag{G.7}$$

- $\mathit{failLeqSize}(AUX, D, R) \quad \doteq$

$$\mathit{failGeqSize}(AUX, R, D) \tag{G.8}$$

- $\mathit{failEqSize}(AUX, D, R) \quad \doteq$

$$\mathit{failLeqSize}(AUX, D, R) \cup \mathit{failGeqSize}(AUX, D, R) \tag{G.9}$$

  where AUX is an auxiliary guessed relation that encodes the function between D and R, and will be omitted as an argument if it is not used anywhere else.

G.2 Proofs of results

Proof of Theorem 8.4.3.
Only if part. If $D \in \vec{D}$, it follows that an extension $\vec{\Sigma}$ for predicates in $\vec{S}$ exists, such that:

$$[D, \vec{\Sigma}] \models \forall \vec{X}\ \exists \vec{Y}\ \varphi(\vec{X}, \vec{Y}).$$

By translating φ on the right side into relational algebra (according to the point 1 in Section 8.4.2), we obtain a relational expression *PHI* on the relational vocabulary given by relations corresponding to predicates in $\vec{R}$, plus those corresponding to predicates in $\vec{S}$ (i.e., relations in $\vec{Q}$).

When evaluating *PHI* on the new database $[D, \vec{\Xi}]$, where $\vec{\Xi}$ are the extensions of relations in $\vec{Q}$ corresponding to the extensions of predicates in $\vec{\Sigma}$, we obtain that for all tuples $\langle \vec{X} \rangle$ there exists a tuple $\langle \vec{Y} \rangle$ such that the tuple $\langle \vec{X}, \vec{Y} \rangle$ belongs to *PHI*, i.e.:

$$\forall \langle \vec{X} \rangle \exists \langle \vec{Y} \rangle : \langle \vec{X}, \vec{Y} \rangle \in \mathit{PHI}.$$

Since $\langle \vec{X} \rangle \in \mathit{DOM}^{|\vec{X}|}$, we obtain that $\mathit{DOM}^{|\vec{X}|} \subseteq \underset{\vec{X}}{\pi}(\mathit{PHI})$, implying that the expression for *FAIL* in the *NP-Alg* query (8.8) evaluates to the empty relation for the extension $\vec{\Xi}$ of the guessed tables $\vec{Q}$.

If part. Suppose that $D \notin \vec{D}$. This implies that

$$D \models \neg \exists \vec{S}\ \forall \vec{X}\ \exists \vec{Y}\ \varphi(\vec{X}, \vec{Y}$$

or, equivalently, that:

$$D \models \forall \vec{S}\ \exists \vec{X}\ \forall \vec{Y}\ \neg \varphi(\vec{X}, \vec{Y})$$

By translating formula φ into relational algebra, we obtain that, for every extension $\vec{\Xi}$ of relations in $\vec{Q}$ (corresponding to predicates in $\vec{S}$), there exists at least one tuple $\langle\vec{X}\rangle$ such that for every tuple $\langle\vec{Y}\rangle$, tuple $\langle\vec{X},\vec{Y}\rangle$ does not belong to *PHI*. This implies that:

$$\exists\langle\vec{X}\rangle \in DOM^{|\vec{X}|} : \langle\vec{X}\rangle \notin \underset{\vec{X}}{\pi}(PHI),$$

and so that the expression for *FAIL* in the *NP-Alg* query (8.8) does not evaluate to the empty relation for all possible extensions $\vec{\Xi}$ of the guessed tables $\vec{Q}$. □

G.2.1 Combined complexity of *NP-Alg*

In this section we prove Theorem 8.4.2. The proof consists in reducing an NE-complete problem, *Succint* 3*-coloring* (82), i.e., the "succinct version" of the graph 3-coloring problem, into an *NP-Alg* query. The Succint 3-coloring problem is defined as follows:

Definition G.2.1 (The Succint 3-coloring problem). *Nodes of the input graph are elements of* $\{0,1\}^n$*, and, instead of an explicitly given EDGES relation, there is a* boolean circuit *with* $2n$ *inputs and one output, such that the value output by the circuit is 1 if and only if the inputs are two* n*-tuples that encode a pair of nodes connected by an edge. A boolean circuit is a finite set of triples* $\{(a_i, b_i, c_i), i = 1, \ldots, k\}$*, where* $a_i \in \{OR, AND, NOT, IN\}$ *is the kind of the gate, and* $b_i, c_i < i$ *are the inputs of the gate (hence, the whole circuit is acyclic), unless the gate is an input gate (*$a_i = IN$*), in which case, say,* $b_i = c_i = 0$*. For NOT gates,* $b_i = c_i$*. Given values in* $\{0,1\}$ *for the input gates, we can compute the values of all gates one by one by starting from the first one. The value of the circuit is the value of the last gate. Finally, the* Succint 3-coloring problem *is the following: Given a boolean circuit with* $2n$ *inputs and one output, is the graph thus presented* 3*-colorable?*

The Succint 3-coloring problem is proven to be NE-complete in the same paper (82).

Reduction of Succint 3-coloring into an *NP-Alg* query. Given an input boolean circuit $G = \{g_i = (a_i, b_i, c_i) \mid 1 \le i \le k\}$ with $2n$ inputs and one output, we construct the *NP-Alg* query ψ that specifies the Succint 3-coloring problem (on the graph represented by circuit G) as follows.

As for the set $\vec{Q}$ of guessed relations, we declare a relation $G_i^{(2n)}$ for every gate i, i.e., for every triple $g_i = (a_i, b_i, c_i)$, $(1 \le i \le k)$. Moreover, we declare in $\vec{Q}$ three more relations, $COL_1^{(n)}$, $COL_2^{(n)}$, $COL_3^{(n)}$, encoding the partition of the nodes into 3 groups, analogously to the specification for k-coloring given in Example 8.3.1. So, the *Guess* part of the *NP-Alg* query being built is the following:

$$\textit{Guess } G_1^{(2n)}, \ldots, G_k^{(2n)}, COL_1^{(n)}, COL_2^{(n)}, COL_3^{(n)};$$

Intuitively, relations G_i will contain all tuples $\langle\vec{X},\vec{Y}\rangle$, with $\vec{X} = \langle X_1, \ldots, X_n\rangle$, and $\vec{Y} = \langle Y_1, \ldots, Y_n\rangle$ (i.e., binary encodings of the nodes X and Y) for all pairs of nodes X and Y that make the output of the i-th gate 1.

The expression for *FAIL* is of the following kind:

$$FAIL = FAIL\_CIRCUIT \cup FAIL\_PARTITION \cup FAIL\_COLORING.$$

The first subexpression evaluates to the empty relation if and only if the guessed extension for the G_i relations correctly encodes the circuit, while the second and the third ones evaluate to the empty relation if and only if relations COL_1, COL_2, COL_3, are a partition of the graph nodes and a correct coloring of the graph (we omit their definitions, since they are very similar to those presented in Example 8.3.1).

The expression for *FAIL_CIRCUIT* contains in turn one of the following subexpressions $FAIL\_G_i$, $1 \leq i \leq k$, for every gate i, according to its type a_i. In particular:

- If $a_i = AND$, then $FAIL\_G_i = G_i \ \Delta \ \left[G_{b_i} \ \cap \ G_{c_i}\right]$;
- If $a_i = OR$, then $FAIL\_G_i = G_i \ \Delta \ \left[G_{b_i} \ \cup \ G_{c_i}\right]$;
- If $a_i = NOT$, then $FAIL\_G_i = G_i \ \Delta \ \left[DOM_{01}^{2n} - G_{b_i}\right]$;
- If $a_i = IN$, then $FAIL\_G_i = G_i \ \Delta \ \underset{\$j=1}{\sigma}(DOM_{01}^{2n})$, assuming that the i-th gate (of type IN) is the j-th input of the circuit.

In the above definition, we used of the relation DOM_{01}, defined as:

$$DOM_{01} = \underset{\substack{\$1 \neq AND \ \wedge \\ \$1 \neq OR \ \wedge \\ \$1 \neq NOT \ \wedge \\ \$1 \neq IN}}{\sigma}(DOM)$$

that will contain at most the two tuples $\langle 0 \rangle$ and $\langle 1 \rangle$ (since DOM would also contain constants for the gate types). Thus, the expression for *FAIL_CIRCUIT* is the following:

$$FAIL\_CIRCUIT \ = \ \bigcup_{i=1}^{k} FAIL\_G_i.$$

It remains to prove that the expression for *FAIL_CIRCUIT* evaluates to the empty relation if and only if guessed relations $G_1, \ldots, G_k$ correctly encode the boolean circuit representing the input graph, i.e., if and only if for all i, relation G_i contains exactly all $2n$-tuples (encoding pairs of nodes given as input to the circuit) that make the output of the i-th gate 1. This is what the following lemma claims.

Lemma G.2.1. *Let $G = \{g_i = (a_i, b_i, c_i) \mid 1 \leq i \leq k\}$ be a boolean circuit encoding a graph, and let ψ be the NP-Alg query built as described above. An extension for guessed tables $G_1^{(2n)}, \ldots, G_k^{(2n)}$ exists such that the expression for FAIL_CIRCUIT evaluates to the empty relation. Moreover, for such an extension, each G_i contains exactly all $2n$-tuples $\langle X_1, \ldots, X_n, Y_1, \ldots, Y_n\rangle$ that make the output of the i-th gate 1. As a consequence, the extension for G_k contains all $2n$-tuples that encode pairs of nodes linked by an edge.*

Proof. We first show that, if extensions for $G_1, \ldots G_k$ in the *NP-Alg* query ψ exist that make the expression for *FAIL_CIRCUIT* evaluate to the empty relation (by making all the expressions for $FAIL\_G_i$ evaluate to the empty relation), then, for every input $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ to the circuit, each gate i ($1 \leq i \leq k$) outputs 1 if and only if the $2n$-tuple $\langle X_1, \ldots, X_n, Y_1, \ldots, Y_n\rangle$ belongs to the corresponding G_i. Secondly, we show that such an extension indeed exists. The proof of the first point is by induction on the index i:

$i = 1$: Gate g_1 is, by construction, of type IN (i.e., $a_1 = IN$). Let us assume that g_1 is the j-th input to the circuit, i.e., its output is 1 if and only if the j-th input to the circuit is 1. As it can be observed from the definition of *FAIL_G*$_1$, since by hypothesis it evaluates to the empty relation, G_1 contains all $2n$-tuples that have 1 as the j-th component.

$i > 1$: Let us assume that the lemma holds for all i' such that $1 \leq i' < i$, and let us consider the i-th gate (of type $a_i \in \{IN, AND, OR, NOT\}$) and the extension for the corresponding guessed relation G_i. Since, by hypothesis, *FAIL_G*$_i$ evaluates to the empty relation, it can be observed by its definition that:

- If $a_i = IN$, assuming that g_i is the j-th input to the circuit, G_i contains, by construction, all $2n$-tuples that have 1 as the j-th component;
- If $a_i = AND$, it follows by induction that G_{b_i} and G_{c_i} contain exactly those tuples that make the output of, respectively, gates g_{i_b} e g_{c_i} 1. By construction, the extension for G_i contains exactly those tuples that belong to both G_{b_i} and G_{c_i}.
- If $a_i = OR$ an analogous argument holds, showing that G_i contains exactly those tuples that belong to G_{b_i} or to G_{c_i}.
- If $a_i = NOT$, it follows by induction that G_{b_i} (in this case $b_i = c_i$) contains exactly those tuples that make the output of gate g_{b_i} 1. By construction, the extension for G_i contains exactly those tuples in DOM_{01}^{2n} that do not belong to G_{b_i}.

As for the second point of the proof, it is easy to show that an extension for $G_1, \ldots G_k$ that makes all the expressions for *FAIL_G*$_i$ evaluate to the empty relation indeed exists. The key observation is that expressions for *FAIL_G*$_i$ essentially define which tuples must belong to each G_i (more precisely, each *FAIL_G*$_i$ evaluates to the empty relation if and only if G_i contains exactly the tuples that belong to the relational algebra expression on the right of the "Δ" symbol), and that the *Guess* part of the query generates all possible extensions of those relations with elements in $DOM \supset DOM_{01}$. ☐

Lemma G.2.1 claims that an extension for $G_1, \ldots G_k$ in query ψ that makes the expression for *FAIL_CIRCUIT* evaluate to the empty relation exists, and is the one that correctly models the boolean circuit representing the input graph. It remains to prove that the whole query ψ is such that *FAIL*$\Diamond\emptyset$ if and only if the input graph is 3-colorable. This is claimed by the following result:

Lemma G.2.2. *Let $G = \{g_i = (a_i, b_i, c_i) \mid 1 \leq i \leq k\}$ be a boolean circuit encoding a graph, and let ψ be the NP-Alg query built as described above. The expression for FAIL in ψ evaluates to the empty relation for a given extension of $G_1, \ldots G_k, COL_1, COL_2, COL_3$ if and only if $G_1, \ldots G_k$ correctly encode the circuit G and COL_1, COL_2, COL_3 represent a valid coloring of the input graph. Thus, FAIL$\Diamond\emptyset$ if and only if the input graph is* 3*-colorable.*

Proof. The boolean circuit G is translated into k guessed relations $G_1, \ldots G_k$. The correctness of the translation is claimed by Lemma G.2.1. Moreover, the *Guess* part of query ψ generates also all possible extensions for three more guessed relations, i.e., COL_1, COL_2, COL_3. As discussed in Example 8.3.1, the expression for $\mathit{FAIL\_PARTITION} \cup \mathit{FAIL\_COLORING}$ evaluates to the empty relation if and only if COL_1, COL_2, COL_3 define a valid coloring of the graph. □

From previous lemmas, it follows the proof of Theorem 8.4.2 that states the combined complexity of *NP-Alg*:

Proof of Theorem 8.4.2. Immediate, from Lemma G.2.2 and from the NE-completeness of the Succint 3-coloring problem (82). □

Appendix H

OPL code for the examples

H.1 HP 2D-Protein folding, Example 5.1.1

H.1.1 Model with binary inequalities for non-crossing

```
int+ n = ...;              // Part of the inst. schema: string length

enum Aminoacid {H,P};
range Pos [0..n-1];
range PosButLast [0..n-2];

Aminoacid seq[Pos] = ...;  // Part of the inst. schema: amino-acid seq.

enum Dir {N,E,S,W};
range Coord [-(n-1)..n-1];

// Guessed predicates
var Dir Moves[PosButLast];    // Protein shape
var Coord X[Pos], Y[Pos];     // Abs coordinates for each amino-acid
var Pos contactsNumber;

maximize contactsNumber
subject to {
   contactsNumber = (sum(t1, t2 in Pos : t1+1 < t2)
     (((seq[t1] = H) & (seq[t2] = H)) &
       ((abs(X[t1] - X[t2]) + abs(Y[t1] - Y[t2])) = 1)));

   X[0] = 0; Y[0] = 0;        // Pos. of initial elem. of the seq.
   forall (t in Pos: t > 0) { // Chann. constraints for position
       (Moves[t-1] = N) => (X[t] = X[t-1]     & Y[t] = Y[t-1] + 1);
       (Moves[t-1] = S) => (X[t] = X[t-1]     & Y[t] = Y[t-1] - 1);
       (Moves[t-1] = E) => (X[t] = X[t-1] + 1 & Y[t] = Y[t-1]);
       (Moves[t-1] = W) => (X[t] = X[t-1] - 1 & Y[t] = Y[t-1]);
   };
   // Non-crossing constraint
   forall(t1, t2 in Pos : t2 > t1) {
     (X[t1] <> X[t2]) \/ (Y[t1] <> Y[t2])
   }
};
```

H.1.2 Model with new guessed predicate for non-crossing

```
int+ n = ...;                    // Part of the inst. schema: string length

enum Aminoacid {H,P};
range Pos [0..n-1];
range PosButLast [0..n-2];

Aminoacid seq[Pos] = ...;        // Part of the inst. schema: amino-acid seq.

enum Dir {N,E,S,W};
range Coord [-(n-1)..n-1];
range Hit [0..n/2];              // Number of hits for a cell

// Guessed predicates
var Dir Moves[PosButLast];       // Protein shape
var Coord X[Pos], Y[Pos];        // Abs coordinates for each amino-acid
var Pos contactsNumber;
var Hit Hits[Coord,Coord,Pos];   // Number of times a cell has

maximize contactsNumber
subject to {
   contactsNumber = (sum(t1, t2 in Pos : t1+1 < t2)
     (((seq[t1] = H) & (seq[t2] = H)) &
       ((abs(X[t1] - X[t2]) + abs(Y[t1] - Y[t2])) = 1)));

   X[0] = 0; Y[0] = 0;           // Pos. of initial elem. of the seq.
   forall (t in Pos: t > 0) {    // Chann. constraints for position
       (Moves[t-1] = N) => (X[t] = X[t-1]     & Y[t] = Y[t-1] + 1);
       (Moves[t-1] = S) => (X[t] = X[t-1]     & Y[t] = Y[t-1] - 1);
       (Moves[t-1] = E) => (X[t] = X[t-1] + 1 & Y[t] = Y[t-1]);
       (Moves[t-1] = W) => (X[t] = X[t-1] - 1 & Y[t] = Y[t-1]);
   };

   // Non-crossing: Initially, no cell has been hit ...
   forall (x, y in Coord : x<>0 \/ y<>0) Hits[x,y,0] = 0;
   Hits[0,0,0] = 1;              // ... but the origin

   // Non-crossing: Channelling constraints for Hits
   forall (t in Time, x, y in Coord: t > 0) {
    ( (x = X[t] & y = Y[t]) => (Hits[x,y,t] = Hits[x,y,t-1] + 1) ) &
    ( (not (x = X[t] & y = Y[t])) => Hits[x,y,t] = Hits[x,y,t-1] );
   };
   // Non-crossing: Each cell is hit 0 or 1 times (string does not cross)
   forall (x, y in Coord, t in Time) Hits[x,y,t] <= 1;
};
```

H.2 Sailco inventory, Example 5.3.2

```
int+ nbPeriods = ...;            range Periods 1..nbPeriods;
float+ demand[Periods] = ...;    float+ regularCost = ...;
float+ extraCost = ...;          float+ capacity = ...;
float+ inventory = ...;          float+ inventoryCost = ...;

var float+ regulBoat[Periods];   var float+ extraBoat[Periods];
var float+ inv[0..nbPeriods];

minimize ...                     // Objective function (omitted)
subject to {                     // Constraints
   inv[0] = inventory;
   forall(t in Periods) regulBoat[t] <= capacity;
   forall(t in Periods) regulBoat[t]+extraBoat[t]+inv[t-1] = inv[t]+demand[t];
};
```

H.3 Blocks world, Example 5.3.3

```
int nblocks = ...;
range Block 1..nblocks;              range BlockOrTable 0..nblocks;
BlockOrTable TABLE = 0;              range Time 1..2*nblocks;
range TimeWithZero 0..2*nblocks;     range bool 0..1;
BlockOrTable OnAtStart[Block] = ...; BlockOrTable OnAtGoal[Block] = ...;

// MoveBlock[t] and MoveTo[t] refer to moves performed from time t-1 to time t
var Block MoveBlock[Time];
var BlockOrTable MoveTo[Time];
var BlockOrTable On[Block, TimeWithZero];      // Dependent guessed function
var bool Clear[BlockOrTable, TimeWithZero];    // Dependent guessed function
var TimeWithZero schLen;                       // Schedule length (to minimize)

minimize schLen
subject to {
   forall (b in Block) On[b,0] = OnAtStart[b]; // Initial state (time = 0);
   forall (b in Block, t in TimeWithZero) {    // Chann. constr's for Clear
      ( ( sum(b_up in Block) (On[b_up,t]=b) ) > 0 ) <=> (Clear[b,t] = 0); };
   forall (t in TimeWithZero) { Clear[TABLE,t] = 1; };
   forall (t in Time) {                        // Moves
     (MoveBlock[t] <> MoveTo[t]);
     (MoveTo[t] <> On[MoveBlock[t],t-1]);      // No useless moves
     (t <= schLen) => (
       (Clear[ MoveBlock[t], t-1 ] = 1) &      // Moving block must be clear
       (Clear[ MoveTo[t], t-1 ] = 1 )  &       // Target pos. must be clear
       (On[ MoveBlock[t], t ] = MoveTo[t]) );  // Chann. constr's for On
     forall (b in Block) {                     // Chann. constr's for On
         (t <= schLen) => (                    // (frame conditions)
            (b <> MoveBlock[t]) => (On[b,t] = On[b,t-1]) ) };
   };
   forall (b in Block) { On[b,schLen] = OnAtGoal[b]; }; // Final state
};
```

Bibliography

[1] Aarts, E., and Lenstra, J. K. *Local search in combinatorial optimization.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester, 1997.

[2] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases.* Addison Wesley Publ. Co., Reading, Massachussetts, 1995.

[3] Aho, A. V., Sagiv, Y., and Ullman, J. D. Equivalences among relational expressions. *SIAM Journal on Computing 2*, 8 (1979), 218–246.

[4] Backofen, R., and Will, S. Excluding symmetries in constraint-based search. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)* (Alexandria, VA, USA, 1999), vol. 1713 of *Lecture Notes in Computer Science*, Springer, pp. 73–87.

[5] Bakewell, A., Frisch, A. M., and Miguel, I. Towards automatic modelling of constraint satisfaction problems: A system based on compositional refinement. In *Proceedings of the Second International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)* (Kinsale, Ireland, 2003), pp. 3–17.

[6] Beasley, J. E., Krishnamoorthy, M., Sharaiha, Y. M., and Abramson, D. Scheduling aircraft landings - the static case. *Transportation Science 34* (2000), 180–197.

[7] Ben-Eliyahu, R., and Dechter, R. Default reasoning using classical logic. *Artificial Intelligence 84*, 1–2 (1996), 113–150.

[8] Bibel, W. Constraint satisfaction from a deductive viewpoint. *Artificial Intelligence 35* (1988), 401–413.

[9] Börger, E., Gräedel, E., and Gurevich, Y. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer, 1997.

[10] Borrett, J. E., and Tsang, E. P. K. A context for constraint satisfaction problem formulation selection. *Constraints 4*, 6 (2001), 299–327.

[11] Brown, C. A., Finkelstein, L., and Purdom, P. W. Backtrack searching in the presence of symmetry. In *Proceedings of the Sixth International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting codes* (Rome, Italy, 1988), T. Mora, Ed., vol. 357 of *Lecture Notes in Computer Science*, Springer, pp. 99–110.

[12] Cadoli, M., and Donini, F. M. A survey on Knowledge Compilation. *AI Communications — The European Journal on Artificial Intelligence 10* (1997), 137–150.

[13] Cadoli, M., Donini, F. M., Liberatore, P., and Schaerf, M. Preprocessing of intractable problems. *Information and Computation 176*, 2 (2002), 89–120.

[14] Cadoli, M., Donini, F. M., and Schaerf, M. Is intractability of non-monotonic reasoning a real drawback? *Artificial Intelligence 88*, 1–2 (1996), 215–251.

[15] Cadoli, M., Ianni, G., Palopoli, L., Schaerf, A., and Vasile, D. np-spec: An executable specification language for solving all problems in NP. *Computer Languages 26* (2000), 165–195.

[16] Cadoli, M., and Mancini, T. Combining Relational Algebra, sql, Constraint Modelling, and local search. *Theory and Practice of Logic Programming*, Special Issue on Multiparadigm Languages and Constraint Programming. To appear.

[17] Cadoli, M., and Mancini, T. Combining Relational Algebra, SQL, and Constraint Programming. In *Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS 2002)* (Santa Margherita Ligure, Genova, Italy, 2002), vol. 2309 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 147–161.

[18] Cadoli, M., and Mancini, T. Knowledge compilation = Query rewriting + View synthesis. In *Proceedings of the Twentyfirst ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS 2002)* (Madison, WI, USA, 2002), ACM Press and Addison Wesley, pp. 199–208.

[19] Cadoli, M., and Mancini, T. Detecting and breaking symmetries on specifications. In *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)* (Kinsale, Ireland, 2003).

[20] Cadoli, M., and Mancini, T. Automated reformulation of specifications by safe delay of constraints. In *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2004)* (Whistler, BC, Canada, 2004), AAAI Press/The MIT Press, pp. 388–398.

[21] Cadoli, M., and Mancini, T. Exploiting functional dependencies in declarative problem specifications. In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA 2004)* (Lisbon, Portugal, 2004), vol. 3229 of *Lecture Notes in Artificial Intelligence*, Springer.

[22] Cadoli, M., and Mancini, T. Exploiting functional dependencies in declarative problem specifications. In *Proceedings of the Third International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)* (Toronto, Canada, 2004).

[23] Cadoli, M., and Mancini, T. Using a theorem prover for reasoning on constraint problems. In *Proceedings of the Third International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)* (Toronto, Canada, 2004).

[24] Cadoli, M., and Mancini, T. Using a theorem prover for reasoning on constraint problems. In *Proceedings of the International Workshop on Constraint Programming and Constraints for Verification (CP+CV 2004), in conjunction with the European Joint Conferences on Theory and Practice of Software (ETAPS 2004)* (Barcelona, Spain, 2004).

[25] Cadoli, M., and Palopoli, L. Circumscribing datalog: expressive power and complexity. *Theoretical Computer Science 193* (1998), 215–244.

[26] Cadoli, M., and Schaerf, A. Compiling problem specifications into SAT. *Artificial Intelligence*. To appear.

[27] Castillo, E., Conejo, A. J., Pedregal, P., Garca, R., and Alguacil, N. *Building and Solving Mathematical Programming Models in Engineering and Science*. John Wiley & Sons, 2001.

[28] Chandra, A., and Harel, D. Computable queries for relational databases. *Journal of Computer and System Sciences 21* (1980), 156–178.

[29] Chandra, A. K., and Markowsky, G. On the number of prime implicants. *Discrete Mathematics 24* (1978), 7–11.

[30] Chandra, A. K., and Merlin, P. M. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the Ninth ACM Symposium on Theory of Computing (STOC'77)* (Boulder, CO, USA, 1977), ACM Press, pp. 77–90.

[31] Chang, C. C., and Keisler, H. J. *Model Theory, 3rd ed.* North-Holland, 1990.

[32] Cheeseman, P., Kanefski, B., and Taylor, W. M. Where the really hard problem are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)* (Sydney, Australia, 1991), Morgan Kaufmann, Los Altos, pp. 163–169.

[33] Cheng, B. M. W., Choi, K. M. F., Lee, J. H.-M., and Wu, J. C. K. Increasing constraint propagation by redundant modeling: an experience report. *Constraints 4*, 2 (1999), 167–192.

[34] Choueiry, B. Y., and Noubir, G. On the computation of local interchangeability in discrete constraint satisfaction problems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)* (Madison, WI, USA, 1998), AAAI Press/The MIT Press, pp. 326–333.

[35] Colmerauer, A. Prolog II Manuel de Reference at Modele Theorique, 1982.

[36] Colmerauer, A. Prolog III Reference and Users Manual, version 1.1, 1990.

[37] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* The MIT Press, 1990.

[38] Crawford, J. M., and Baker, A. B. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)* (Seattle, WA, USA, 1994), AAAI Press/The MIT Press, pp. 1092–1097.

[39] Crawford, J. M., Ginsberg, M. L., Luks, E. M., and Roy, A. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96)* (Cambridge, MA, USA, 1996), Morgan Kaufmann, Los Altos, pp. 148–159.

[40] Crawford, J. M., Ginsberg, M. L., Luks, E. M., and Roy, A. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96)* (Cambridge, MA, USA, 1996), Morgan Kaufmann, Los Altos, pp. 148–159.

[41] Crescenzi, P., Goldman, D., Papadimitriou, C. H., Piccolboni, A., and Yannakakis, M. On the complexity of protein folding. *Journal of Computational Biology 5*, 3 (1998), 423–466.

[42] Darwiche, A., and Marquis, P. A perspective on knowledge compilation. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)* (Seattle, WA, USA, 2001), Morgan Kaufmann, Los Altos, pp. 470–475.

[43] Dechter, A., and Dechter, R. Removing redundancies in constraint networks. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87)* (Seattle, WA, USA, 1987), AAAI Press/The MIT Press, pp. 105–109.

[44] Dechter, R. Constraint networks (survey). In *Encyclopedia of Artificial Intelligence, 2nd edition.* John Wiley & Sons, 1992, pp. 276–285.

[45] Di Gaspero, L., and Schaerf, A. EASYLOCAL++: An object-oriented framework for flexible design of local search algorithms. *Software — Practice and Experience 33*, 8 (2003), 733–765.

[46] DIMACS Repository. Available at `ftp://dimacs.rutgers.edu/pub/challenge`.

[47] East, D., and Truszczyński, M. Predicate-calculus based logics for modeling and solving search problems. *ACM Transactions on Computational Logic* (2004). To appear.

[48] Ebbinghaus, H. D., and Flum, J. *Finite Model Theory.* Springer, 1999.

[49] ECL^iPS^e Home page. `www-icparc.doc.ic.ac.uk/eclipse/`.

[50] Eiter, T., and Gottlob, G. Propositional circumscription and extended closed world reasoning are Π_2^p-complete. *Theoretical Computer Science 114* (1993), 231–245.

[51] Fagin, R. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation* (1974), R. M. Karp, Ed., American Mathematical Society, pp. 43–74.

[52] Fahle, T., Schamberger, S., and Sellmann, M. Symmetry breaking. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001)* (Paphos, Cyprus, 2001), vol. 2239 of *Lecture Notes in Computer Science*, Springer, pp. 93–107.

[53] Flener, P. Towards relational modelling of combinatorial optimisation problems. In *Proceedings of the International Workshop on Modelling and Solving Problems with Constraints, in conjunction with the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)* (Seattle, WA, USA, 2001), C. Bessière, Ed.

[54] Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., and Walsh, T. Breaking row and column symmetries in matrix models. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)* (Ithaca, NY, USA, 2002), vol. 2470 of *Lecture Notes in Computer Science*, Springer, p. 462 ff.

[55] Flener, P., Pearson, J., and Ågren, M. Introducing esra, a relational language for modelling combinatorial problems. In *Proceedings of International Symposium LOPSTR 2003: Revised selected papers* (Uppsala, Sweden, 2004), vol. 3018 of *Lecture Notes in Computer Science*, Springer, pp. 214–232.

[56] Focacci, F., and Milano, M. Global cut framework for removing symmetries. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001)* (Paphos, Cyprus, 2001), vol. 2239 of *Lecture Notes in Computer Science*, Springer, pp. 77–92.

[57] Fourer, R., Gay, D. M., and Kernigham, B. W. *AMPL: A Modeling Language for Mathematical Programming.* International Thomson Publishing, 1993.

[58] Freuder, E. C. Modeling: The final frontier. In *Proceedings of the First International Conference on the Practical Application of Constraint Technologies and Logic Programming (PACLP'99)* (London, UK, 1999).

[59] Freuder, E. C., and Sabin, D. Interchangeability supports abstraction and reformulation for multi-dimensional constraint satisfaction. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)* (Providence, RI, USA, 1997), AAAI Press/The MIT Press, pp. 191–196.

[60] Frisch, A., Miguel, I., and Walsh, T. CGRASS: A system for transforming constraint satisfaction problems. In *Proceedings of the Joint Workshop of the ERCIM Working Group on Constraints and the CologNet area on Constraint and Logic Programming on Constraint Solving and Constraint Logic Programming (ERCIM 2002)* (Cork, Ireland, 2002), vol. 2627 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 15–30.

[61] Frisch, A. M., and Peugniez, T. J. Solving non-boolean satisfiability problems with stochastic local search. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001)* (Seattle, WA, USA, 2001), Morgan Kaufmann, Los Altos, pp. 282–290.

[62] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco, CA, USA, 1979.

[63] Gent, I. P., and Smith, B. Symmetry breaking during search in constraint programming. In *Proceedings of the Fourteenth European Conference on Artificial Intelligence (ECAI 2000)* (Berlin, Germany, 2000), pp. 599–603.

[64] Gent, I. P., and Walsh, T. CSPLib: a benchmark library for constraints. Tech. rep., APES-09-1999, 1999. Available from `http://www.csplib.org`.

[65] Giunchiglia, E., and Sebastiani, R. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proceedings of the Sixth Conference of the Italian Association for Artificial Intelligence (AI*IA'99)* (Bologna, Italy, 2000), vol. 1792 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 84–94.

[66] Giunchiglia, F., and Walsh, T. A theory of abstraction. *Artificial Intelligence 57* (1992), 323–389.

[67] Glover, F., and Laguna, M. *Tabu search.* Kluwer Academic Publisher, Boston, USA, 1997.

[68] Gogic, G., Kautz, H. A., Papadimitriou, C., and Selman, B. The comparative linguistics of knowledge representation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)* (Montral, QC, Canada, 1995), Morgan Kaufmann, Los Altos, pp. 862–869.

[69] Gottlob, G., and Kolatis, P., and Schwentick, T. Existential second-order logic over graphs: Charting the tractability frontier. In *Proceedings of the Forty-first Annual Symposium on the Foundations of Computer Science (FOCS 2000)* (Redondo Beach, CA, USA, 2000), IEEE Computer Society Press.

[70] Hart, W., and Istrail, S. HP Benchmarks. Available at `http://www.cs.sandia.gov/tech_reports/compbio/tortilla-hp-benchmarks.html`.

[71] Hnich, T., and Walsh, T. Why Channel? Multiple viewpoints for branching heuristics. In *Proceedings of the Second International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)* (Kinsale, Ireland, 2003).

[72] ILOG AMPL CPLEX system version 7.0 user's guide. Available at `www.ilog.com`, 2001.

[73] ILOG optimization suite — white paper. Available at `www.ilog.com`, 1998.

[74] ILOG DBLink 4.1 Tutorial. Available at `www.ilog.com`, 1999.

[75] Jaffar, J., and Lassez, J.-L. Constraint logic programming. In *Proceedings of the Fourteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)* (Munich, Germany, 1987), ACM Press, pp. 111–119.

[76] Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. The CLP(R) Language and System. *ACM Transactions on Programming Languages 14*, 3 (1992), 339–395.

[77] Karp, R. M., and Lipton, R. J. Some connections between non-uniform and uniform complexity classes. In *Proceedings of the Twelfth ACM Symposium on Theory of Computing (STOC'80)* (1980), ACM Press, pp. 302–309.

[78] Kautz, H., and Selman, B. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)* (Portland, OR, USA, 1996), AAAI Press/The MIT Press, pp. 1194–1201.

[79] Kautz, H. A., and Selman, B. Forming concepts for fast inference. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)* (San Jose, CA, USA, 1992), AAAI Press/The MIT Press, pp. 786–793.

[80] Klug, A. On conjunctive queries containing inequalities. *Journal of the ACM 1*, 35 (1988), 146–160.

[81] Kolaitis, P. G. Constraint satisfaction, databases, and logic. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)* (Acapulco, Mexico, 2003), Morgan Kaufmann, Los Altos, pp. 1587–1595.

[82] Kolaitis, P. G., and Papadimitriou, C. H. Why not negation by fixpoint? *Journal of Computer and System Sciences 43* (1991), 125–144.

[83] Kolaitis, P. G., and Thakur, M. N. Logical definability of NP optimization problems. *Information and Computation 115*, 2 (1994), 321–353.

[84] Lau, K. F., and Dill, K. A. A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *Macromolecules 22* (1989), 3986–3997.

[85] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*. To appear.

[86] Li, C. M. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)* (Austin, TX, USA, 2000), AAAI Press/The MIT Press.

[87] Li, C. M., and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)* (Nagoya, Japan, 1997), Morgan Kaufmann, Los Altos, pp. 366–371.

[88] Lifschitz, V. Computing circumscription. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI'85)* (Los Angeles, CA, USA, 1985), Morgan Kaufmann, Los Altos, pp. 121–127.

[89] M. Dincbas, P. Van Hentenryck, H. S., and Aggoun, A. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)* (Tokyo, Japan, 1988), Springer, pp. 249–264.

[90] Mancini, T. Reformulation techniques for a class of permutation problems. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)* (Kinsale, Ireland, 2003), vol. 2833 of *Lecture Notes in Computer Science*, Springer, p. 984.

[91] Martello, S., and Toth, P. *Knapsack Problems: algorithms and computer implementation.* John Wiley & Sons, 1990.

[92] McCune, W. MACE 2.0 reference manual and guide. Tech. Rep. ANL/MCS-TM-249, Argonne National Laboratory, Mathematics and Computer Science Division, May 2001. Available at `http://www-unix.mcs.anl.gov/AR/mace/`.

[93] McCune, W. Otter 3.3 reference manual. Tech. Rep. ANL/MCS-TM-263, Argonne National Laboratory, Mathematics and Computer Science Division, August 2003. Available at `http://www-unix.mcs.anl.gov/AR/otter/`.

[94] Meseguer, P., and Torras, C. Solving strategies for highly symmetric CSPs. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)* (Stockholm, Sweden, 1999), Morgan Kaufmann, Los Altos, pp. 400–405.

[95] Meseguer, P., and Torras, C. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence 129* (2001), 133–163.

[96] Michel, L., and Van Hentenryck, P. Localizer. *Constraints 5*, 1 (2000), 43–84.

[97] Miller, D. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation 1*, 4 (1991), 497–536.

[98] Minker, J. On indefinite databases and the closed world assumption. In *Proceedings of the Sixth International Conference on Automated Deduction (CADE'82)* (New York, NY, USA, 1982), vol. 138 of *Lecture Notes in Computer Science*, Springer, pp. 292–308.

[99] Moses, Y., and Tennenholtz, M. Off-line reasoning for on-line efficiency: knowledge bases. *Artificial Intelligence 83* (1996), 229–239.

[100] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the Thirtyeighth Conference on Design Automation (DAC 2001)* (Las Vegas, NV, USA, 2001), ACM Press, pp. 530–535.

[101] Nerode, A., Ng, R. T., and Subrahmanian, V. S. Computing circumscriptive databases. I: Theory and algorithms. *Information and Computation 116* (1995), 58–80.

[102] Nilsson, N. J. *Principles of Artificial Intelligence.* Tioga Publishing Co., 1980.

[103] Older, W., and Benhamou, F. Programming in CLP(BNR). In *Position Papers for the First Workshop on Principles and Practice of Constraint Programming* (Newport, RI, USA, 1993), pp. 239–249.

[104] OR Library. Available at `www.ms.ic.ac.uk/info.html`.

[105] Papadimitriou, C. H. *Computational Complexity.* Addison Wesley Publ. Co., Reading, Massachussetts, Reading, MA, 1994.

[106] Prestwich, S. Supersymmetric modeling for local search. In *Proceedings of the Second International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)* (Ithaca, NY, USA, 2002).

[107] Puget, J.-F. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of the Seventh International Symposium on Methodologies for Intelligent Systems (ISMIS'93)* (Trondheim, Norway, 1993), H. J. Komorowski and Z. W. Ras, Eds., vol. 689 of *Lecture Notes in Computer Science*, Springer, pp. 350–361.

[108] Puget, J.-F. Symmetry breaking revisited. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002)* (Ithaca, NY, USA, 2002), vol. 2470 of *Lecture Notes in Computer Science*, Springer, pp. 446–461.

[109] Riazanov, A., and Voronkov, A. Vampire. In *Proceedings of the Sixteenth International Conference on Automated Deduction (CADE'99)* (Trento, Italy, 1999), vol. 1632 of *Lecture Notes in Computer Science*, Springer.

[110] Rothberg, E. Using cuts to remove symmetry. In *Proceedings of the Seventeenth International Symposium on Mathematical Programming (ISMP 2000)* (Atlanta, GA, USA, 2000).

[111] Sagiv, Y., and Yannakakis, M. Equivalence among relational expressions with the union and difference operations. *Journal of the ACM 4*, 27 (1980), 633–655.

[112] Schaerf, A. A survey of automated timetabling. *Artificial Intelligence Review 13*, 2 (1999), 87–127.

[113] Schaerf, A., Cadoli, M., and Lenzerini, M. LOCAL++: A C++ framework for local search algorithms. *Software — Practice and Experience 30*, 3 (2000), 233–257.

[114] Selman, B., and Kautz, H. A. Knowledge compilation and theory approximation. *Journal of the ACM 43* (1996), 193–224.

[115] Selman, B., Kautz, H. A., and Cohen, B. Local search strategies for satisfiability testing. In *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability* (Providence, RI, USA, 1993), M. Trick and D. S. Johnson, Eds.

[116] Sherali, H. D., and Cole Smith, J. Improving discrete model representations via symmetry considerations. *Management Science 47* (2001), 1396–1407.

[117] SICStus Prolog home page. `http://www.sics.se/sicstus/`.

[118] Simons, P., Niemelä, I., and Soininen, T. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 1–2 (2002), 181–234.

[119] Smith, B. Reducing symmetry in a combinatorial design problem. In *Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2001)* (Ashford, Kent, UK, 2001), pp. 351–360.

[120] Smith, B. M. Dual model of permutation problems. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001)* (Paphos, Cyprus, 2001), vol. 2239 of *Lecture Notes in Computer Science*, Springer, pp. 615–619.

[121] Smith, B. M., Stergiou, K., and Walsh, T. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000)* (Austin, TX, USA, 2000), AAAI Press/The MIT Press, pp. 182–187.

[122] Tsang, E. P. K., Mills, P., Williams, R., Ford, J., and Borrett, J. A computer aided constraint programming system. In *Proceedings of the First International Conference on the Practical Application of Constraint Technologies and Logic Programming (PACLP'99)* (London, UK, 1999), pp. 81–93.

[123] Ullman, J. D. *Principles of Database and Knowledge Base Systems*, vol. 1. Computer Science Press, 1988.

[124] Van Hentenryck, P. *The OPL Optimization Programming Language.* The MIT Press, 1999.

[125] Van Hentenryck, P., Flener, P., Pearson, J., and Ågren, M. Tractable symmetry breaking for CSPs with interchangeable values. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)* (Acapulco, Mexico, 2003), Morgan Kaufmann, Los Altos, pp. 277–282.

[126] Van Hentenryck, P., Flener, P., Pearson, J., and Ågren, M. Compositional derivation of symmetries for constraint satisfaction. Tech. Rep. 2004-022, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2004.

[127] Van Hentenryck, P., and Michel, L. Control abstractions for local search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003)* (Kinsale, Ireland, 2003), vol. 2833 of *Lecture Notes in Computer Science*, Springer, pp. 65–80.

[128] Vardi, M. Y. The complexity of relational query languages. In *Proceedings of the Fourteenth ACM Symposium on Theory of Computing (STOC'82)* (1982), ACM Press, pp. 137–146.

[129] Walsh, T. Permutation problems and channelling constraints. In *Proceedings of the Eighth International Conference on Logic for Programming and Automated Reasoning (LPAR 2001)* (Havana, Cuba, 2001), R. Nieuwenhuis and A. Voronkov, Eds., vol. 2250 of *Lecture Notes in Computer Science*, Springer, pp. 377–391.

[130] Warren, D. H. D. Extract from Kluzniak and Szapowicz APIC studies in data processing, no. 24, 1974. In *Readings in Planning.* Morgan Kaufmann, Los Altos, 1990, pp. 140–153.

[131] Weigel, R., and Bliek, C. On reformulation of constraint satisfaction problems. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI'98)* (Brighton, UK, 1998), John Wiley & Sons, pp. 254–258.

[132] Winslett, M. *Updating Logical Databases.* Cambridge University Press, 1990.

[133] Zhang, W., Rangan, A., and Looks, M. Backbone guided local search for maximum satisfiability. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003)* (Acapulco, Mexico, 2003), Morgan Kaufmann, Los Altos, pp. 1179–1186.

Index