Automated Reformulation of Specifications by Safe Delay of Constraints

Marco Cadoli and **Toni Mancini**
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
cadoli|tmancini@dis.uniroma1.it

Abstract

In this paper we propose a form of reasoning on *specifications* of combinatorial problems, with the goal of reformulating them so that they are more efficiently solvable. The reformulation technique highlights constraints that can be safely "delayed", and solved afterwards. Our main contribution is the characterization (with soundness proof) of safe-delay constraints with respect to a criterion on the specification, thus obtaining a mechanism for the automated reformulation of specifications applicable to a great variety of problems, e.g., graph coloring and job-shop scheduling. This is an advancement with respect to the forms of reasoning done by state-of-the-art-systems, which typically just detect linearity of specifications. Another contribution is a preliminary experimentation on the effectiveness of the proposed technique, which reveals promising time savings.

Introduction

State-of-the-art systems and languages for constraint modelling and programming (e.g., AMPL (Fourer, Gay, & Kernigham 1993), OPL (Van Hentenryck 1999), DLV (Eiter *et al.* 1998), SMODELS (Niemelä 1999), NP-SPEC (Cadoli & Schaerf 2001)) clearly separate the *specification* of a problem from its *instances*, adopting the two-level general architecture depicted in Figure 1. Some of them (e.g., AMPL) also allow the user to choose *a posteriori* one out of several solvers, being able to translate a specification into different formats. Others (e.g., OPL) go one step further, by automatically choosing the most appropriate solver for a problem, thus offering a (primitive) form of *reasoning* on the specification (OPL only checks whether the specification is linear, in this case invoking a linear –typically more efficient– solver).

We aim to a more ambitious long-term goal, i.e., to *automatically reformulate* the specification –independently on the instance– to improve the efficiency of computation. We get inspiration from the relational database technology, since it is well-known that reformulating queries –independently on the database– may result in greater efficiency. As an example, making selections as soon as possible is a simple heuristic that typically allows to decrease the number of accesses to disk (cf., e.g., (Abiteboul, Hull, & Vianu 1995)).

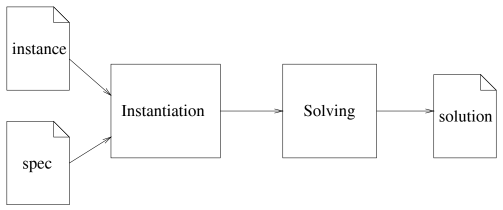

Figure 1: Two-level general architecture for constraint solving.

Reformulation is a difficult task in general: 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 (Börger, Gräedel, & Gurevich 1997). For this reason, our current research focuses on restricted forms of reformulation, and, more specifically, on selecting constraints that can be safely "delayed", and solved afterwards.

The NP-complete graph k-coloring problem offers a simple example of a constraint of this kind. 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). 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 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 which are safe-delay may lead to several advantages:

- The instantiation phase (cf. Figure 1) will typically be faster, since safe-delay constraints are not taken into account. As an example, let's assume we want to use a SAT solver (after instantiation) for the solution of k-coloring

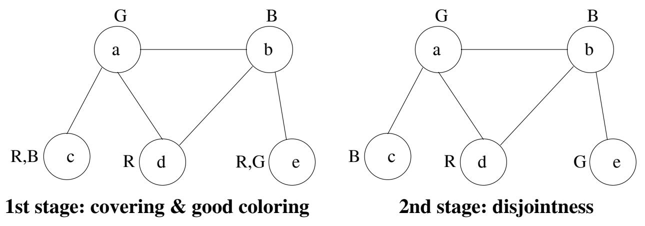

Figure 2: Delaying the disjointness constraint in 3-coloring.

on a graph with n nodes and e edges. The SAT instance encoding the k-coloring instance –in the obvious way, cf., e.g., (Frisch & Peugniez 2001)– 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 (even if preliminary) experiments, using a SAT solver, we obtained a fairly consistent (in some cases, more than one order of magnitude) speed-up for hard instances of various problems, e.g., graph coloring and job-shop scheduling. On top of that, we implicitly obtain several good solutions. The approach seems promising for some classes of instances even when state-of-the-art solvers for integer linear programming like CPLEX are used (cf. Section "Experimental results").

- 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 3 and can be applied to any system which separates the instance from the specification. It is in some sense similar to the well-known *divide and conquer* technique, but rather than dividing the instance, we divide the constraints. In general, the first stage will be more computationally expensive than the second one, which, in our proposal, will always be doable in polynomial time.

The goal of this paper 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.

After recalling some preliminaries, we present our reformulation technique and an experimentation on its effectiveness, on both benchmark and randomly generated instances,

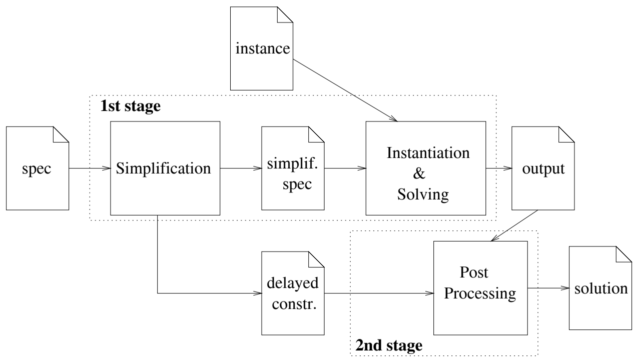

Figure 3: Reformulation architecture.

using both a SAT solver and state-of-the-art linear and constraint programming solvers. We also present a discussion on the adopted experimental methodology. Afterwards, we describe conclusions, and future and related work.

Preliminaries

In this paper, we use *existential second-order logic* (ESO) for the specification of problems, which allows to represent all search problems in the complexity class NP (Fagin 1974). The use of ESO as a modelling language for problem specifications is common in the database literature, but unusual in constraint programming, therefore few comments are in order. Constraint modelling systems like those mentioned in the Introduction have a richer syntax and more complex constructs, and we plan to eventually move from ESO to such languages. For the moment, we claim that studying the simplified scenario is a mandatory starting point for more complex investigations, and that our results can serve as a basis for reformulating specifications written in higher-level languages. In Section "Methodological discussion" we discuss further our choice. Coherently with all state-of-the-art systems, we represent an instance of a problem by means of a *relational database*. All constants appearing in a database are *uninterpreted*, i.e., they don't have a specific meaning.

An ESO specification describing a search problem π is a formula

$$\psi_\pi \doteq \exists \vec{S}\, \phi(\vec{S}, \vec{R}), \tag{1}$$

where $\vec{R} = \{R_1, \ldots, R_k\}$ is the input relational schema (i.e., a fixed set of relations of given arities denoting the schema for all input instances for π), and ϕ is a closed first-order formula on the relational vocabulary $\vec{S} \cup \vec{R} \cup \{=\}$ ("=" is always interpreted as identity).

An instance $\mathcal{I}$ of the problem is given as a relational database over the schema $\vec{R}$, i.e., as an extension for all relations in $\vec{R}$. Predicates (of given arities) in the set $\vec{S} = \{S_1, \ldots, S_n\}$ are called *guessed*, and their possible extensions (with tuples on the domain given by constants occurring in $\mathcal{I}$ plus those occurring in ϕ, i.e., the so called Herbrand universe) encode points in the search space for problem π on instance $\mathcal{I}$.

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

For each instance $\mathcal{I}$:

$$\Sigma \text{ is a solution to } \pi(\mathcal{I}) \iff \{\Sigma, \mathcal{I}\} \models \phi.$$

It is worthwhile to note that, when a specification is instantiated against an input database, a constraint satisfaction problem (in the sense of (Dechter 1992)) is obtained.

Example 1. *In the "three-coloring" NP-complete decision problem (cf. (Garey & Johnson 1979, Prob. GT4)) the input is a graph, and the question is whether it is possible to give each of its nodes one out of three colors (red, green, and blue), in such a way that adjacent nodes (not including self-loops) are never colored the same way. The question can be easily specified as an ESO formula ψ over a binary relation edge:*

$$\begin{aligned}
\exists RGB\ \forall X \quad & R(X) \vee G(X) \vee B(X)\ \wedge && (2)\\
\forall X \quad & R(X) \rightarrow \neg G(X)\ \wedge && (3)\\
\forall X \quad & R(X) \rightarrow \neg B(X)\ \wedge && (4)\\
\forall X \quad & B(X) \rightarrow \neg G(X)\ \wedge && (5)\\
\forall XY \quad & X \neq Y \wedge R(X) \wedge R(Y) \rightarrow \neg edge(X,Y) \wedge && (6)\\
\forall XY \quad & X \neq Y \wedge G(X) \wedge G(Y) \rightarrow \neg edge(X,Y) \wedge && (7)\\
\forall XY \quad & X \neq Y \wedge B(X) \wedge B(Y) \rightarrow \neg edge(X,Y), && (8)
\end{aligned}$$

where clauses (2), (3-5), and (6-8) represent the covering, disjointness, and good coloring constraints, respectively. Referring to the graph in Figure 2, 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).

In what follows, the set of tuples from the Herbrand universe taken by guessed predicates will be called their *extension* and denoted with $ext()$. By referring to the previous example, formula ψ is satisfied, e.g., for $ext(R) = \{d\}$, $ext(G) = \{a, e\}$, $ext(B) = \{b, c\}$ (cf. Figure 2, right). 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.

Reformulation

In this section we show sufficient conditions for constraints of a specification to be safe-delay. We refer to the architecture of Figure 3, with some general assumptions:

1. As shown in Figure 2, the output of the first stage of computation may –implicitly– contain *several* solutions. In the second stage we do not want to compute all of them, but just to arbitrarily select one.
2. The second stage of computation can only *shrink* the extension of a guessed predicate. Figure 4 represents the extensions of the red predicate in the first (R^*) and second (R) stages of Figure 2 ($ext(B)$ 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.

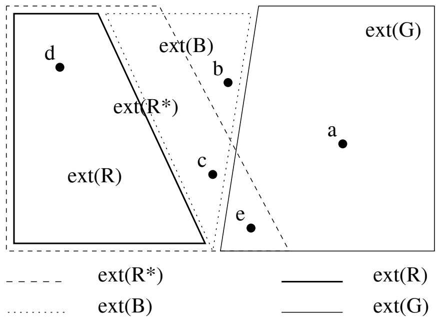

Figure 4: Extensions for the 3-coloring specification.

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, we will initially focus our attention on *a single monadic* guessed predicate, trying to figure out which constraints concerning it can be delayed. Afterwards, we extend our results to *sets* of monadic guessed predicates, then to *binary* predicates.

Single monadic predicate. We refer to the 3-coloring specification of Example 1, focusing on one of the guessed predicates, R, and trying to find an intuitive explanation for the fact that clauses (3–4) can be delayed. We immediately note that clauses in the specification can be partitioned into three subsets NO_R, NEG_R, 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. We note that this is just a possibility, and we are not obliged to do that: as an example, clauses (3–4) 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 (2) is. In fact, if we equivalently rewrite clauses (2) and (3–4), respectively, as follows:

$$\begin{aligned}
\forall X \quad & \neg B(X) \wedge \neg G(X) \rightarrow R(X) && (2)'\\
\forall X \quad & R(X) \rightarrow \neg B(X) \wedge \neg G(X), && (3-4)'
\end{aligned}$$

we note that clause $(2)'$ sets a lower bound for the extension of R, and clauses $(3–4)'$ set an upper bound for it; both the lower and the upper bound are $ext(\neg B(X) \wedge \neg G(X))$. If we use –in the first stage– clauses (2,5–8) for computing $ext(R^*)$ (in place of $ext(R)$), then –in the second stage– we can safely define $ext(R)$ as $ext(R^*) \cap ext(\neg B(X) \wedge \neg G(X))$, and no constraint will be violated (cf. Figure 4). The next theorem shows that is not by chance that the antecedent of $(2)'$ is semantically related to the consequence of $(3–4)'$.

Theorem 1. *Let ψ be an ESO formula of the form:*

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

where Ξ is a conjunction of clauses, both α and β are arbitrary formulae in which R does not occur and X is the only free variable, and it holds that:

Hyp 1: *R either does not occur or occurs negatively in Ξ;*
Hyp 2: $\models \forall X\, \alpha(X) \rightarrow \beta(X)$.

Let ψ^s be:

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

where R^* is a new predicate symbol, and Ξ^* is Ξ with R replaced by R^*, and ψ^d be:

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

For each database D and each list M^s of extensions for $(S_1, \ldots, S_h, R^*)$ such that $(D, M^s) \models \psi^s$, then:

$$(D, M^s - ext(R^*), ext(R)) \models \psi.$$

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

Proof. Let D be any database, M^s be any list of extensions for $(S_1, \ldots, S_h, R^*)$ such that $(D, M^s) \models \psi^s$, and $ext(R)$ be an extension for R such that $(D, M^s, ext(R)) \models \psi^d$.

From the definition of ψ^d, it follows that:

$$(D, M^s, ext(R)) \models \forall X\, R(X) \rightarrow R^*(X),$$

and so, since clauses in Ξ^* contain at most negative occurrences of R^*, that:

$$(D, M^s, ext(R)) \models \Xi. \tag{9}$$

Furthermore, from the definition of ψ^s it follows that:

$$(D, M^s) \models \forall X\, \alpha(X) \rightarrow R^*(X),$$

and from **Hyp 2** that:

$$(D, M^s) \models \forall X\, \alpha(X) \rightarrow R^*(X) \wedge \beta(X).$$

This implies, by the definition of ψ^d, that:

$$(D, M^s, ext(R)) \models \forall X\, \alpha(X) \rightarrow R(X). \tag{10}$$

Moreover, by the same definition, it is also true that:

$$(D, M^s, ext(R)) \models \forall X\, R(X) \rightarrow \beta(X). \tag{11}$$

From (9–11), and from the observation that R^* does not occur in any of the right parts of them, the thesis follows. □

Referring to Figure 3, ψ is the specification, D is the instance, ψ^s is the "simplified specification", and $\forall X\, R(X) \rightarrow \beta(X)$ is the "delayed constraint". Solving ψ^s against D produces –if the instance is satisfiable– a list of extensions M^s (the "output"). Evaluating ψ^d against M^s corresponds to the "PostProcessing" phase in the second stage; 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., (Abiteboul, Hull, & Vianu 1995)), thus in polynomial time.

In other words, the theorem says that, for each satisfiable instance D of the simplified specification ψ^s, each solution M^s of ψ^s can be translated, via ψ^d, to a solution of the original specification ψ; we can also say that $\Xi \wedge \forall X\, \alpha(X) \rightarrow R(X)$ is safe-forget, and $\forall X\, R(X) \rightarrow \beta(X)$ is safe-delay.

Referring to the specification of Example 1, Ξ is the conjunction of clauses (5–8), and $\alpha(X)$ and $\beta(X)$ are both $\neg B(X) \wedge \neg G(X)$, cf. clauses (3–4)$'$. Figure 4 represents possible extensions of the red predicate in the first (R^*) and second (R) stages, for the instance of Figure 2, and Figure 6 (left) shows that, if **Hyp 2** holds, the constraint $\forall X\, \alpha(X) \rightarrow R(X)$ 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 proposition.

Proposition 1. *Let D, ψ, ψ^s and ψ^d as in Theorem 1. For each database D, if ψ is satisfiable, ψ^s and ψ^d are satisfiable.*

Proof. Let D be any database, and M be any list of extensions for $(S_1, \ldots, S_h, R)$ such that $(D, M) \models \psi$. Let R^* be defined in such a way that $ext(R^*) = ext(R)$.

It is immediate to show that $(M - ext(R), ext(R^*)) \models \psi^s$, and $(M, ext(R^*)) \models \psi^d$. □

To substantiate the reasonableness of the two hypotheses of Theorem 1, we play the devil's advocate and add to the specification of Example 1 the constraint

$$\forall X\, edge(X, X) \rightarrow R(X), \tag{12}$$

saying that self-loops must be red. We immediately notice that now clauses (3–4) are not safe-delay: intuitively, after the first stage, nodes may be red either because of (2) or because of (12), and (3–4) are not enough to set the correct color for a node. Now, if –on top of (5–8)– Ξ contains also the constraint (12), **Hyp 1** is clearly not satisfied. Analogously, if (12) 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 6, right, gives further evidence that the constraint $\forall X\, \alpha(X) \rightarrow R(X)$ can be violated if $ext(R)$ is computed using ψ^d and $ext(\alpha)$ is not a subset of $ext(\beta)$.

Summing up, a constraint with a positive occurrence of R can be safely forgotten only if there is a safe-delay constraint which justifies it.

Some further comments about Theorem 1 are in order:

- Ξ does not need to be a conjunction of clauses, but can be any formula such that, from any structure M such that $M \models \Xi$, by shrinking $ext(R)$ and keeping everything

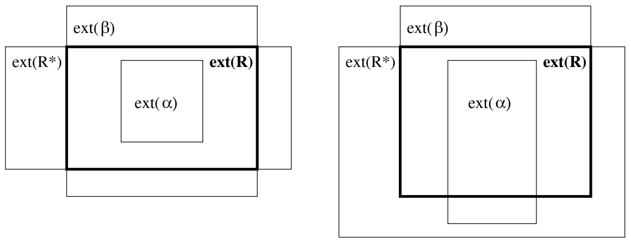

Figure 6: Extensions with and without **Hyp 2**.

else fixed we obtain another model of Ξ. As an example, Ξ may contain the conjunct $\exists X\ R(X) \rightarrow G(X)$.

- 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*.

Set of monadic predicates. Theorem 1 can be applied recursively 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 Proposition 1, 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 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 SOLVE-BYDELAYING 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, we evaluate the procedure REFORMULATE on the specification of Example 1, by focusing on the guessed predicates in the order R, G, B. The output is the following simplified specification $\overline{\psi^s}$ that omits all disjointness constraints (i.e., clauses (3–5)):

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

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

$$\forall X \quad R(X) \leftrightarrow R^*(X) \wedge \neg G(X) \wedge \neg B(X); \tag{13}$$

$$\forall X \quad G(X) \leftrightarrow G^*(X) \wedge \neg B(X). \tag{14}$$

Note 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 (14), and then the extension for R by formula (13) (synthesized, respectively, in the second and first iteration). Since each delayed constraint is first-order, the whole second stage is doable in logarithmic space.

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

Binary predicates. To highlight how our reformulation technique can be extended to handle specifications with binary predicates, we consider the specification of the k-coloring problem using a binary predicate Col –the first argument being the node and the second the color, which is as follows (constraints represent, respectively, covering, disjointness, and good coloring):

$$\begin{array}{lll} \exists Col & \forall X\, \exists Y & Col(X,Y) \wedge \\ & \forall XYZ & Col(X,Y) \wedge Col(X,Z) \rightarrow Y = Z \wedge \\ & \forall XYZ & X \neq Y \wedge Col(X,Z) \wedge Col(Y,Z) \rightarrow \\ & & \neg edge(X,Y). \end{array}$$

Since the number of colors is finite, it is always possible to unfold the above constraints with respect to the second argument of Col. As an example, if $k = 3$, we obtain –up to an appropriate renaming of the Col predicate– the specification of Example 1. The above considerations imply that we can use the architecture of Figure 3 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

$$\begin{array}{l} \exists P \ \ \Xi \ \wedge \ \forall X\, \exists Y\ P(X,Y) \ \wedge \\ \qquad\qquad\qquad \forall XYZ\ \ P(X,Y) \wedge P(X,Z) \rightarrow Y = Z, \end{array}$$

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 1.

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

Example 2. *In the Job-shop scheduling problem (Garey & Johnson 1979, Prob. SS18), 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 performs 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

```
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 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 1, in:
      $\langle\Xi;\ \ \forall X\, \alpha(X) \to R(X);\ \ \forall X\, R(X) \to \beta(X)\rangle$;
    if the prev. step is possible with $\forall X\, \beta(X) \neq$ TRUE
    then begin
      $\overline{\Phi^d}$.push('$\forall X\, R(X) \leftrightarrow R^*(X) \wedge \beta(X)$');
      $\overline{\Phi^s} = \Xi^* \wedge \forall X\ \alpha(X) \to R^*(X)$;
    end;
  end;
  return $\langle\overline{\Phi^s}, \overline{\Phi^d}\rangle$;
end;
```

Figure 5: Algorithm for safe-delay in case of a set of monadic predicates.

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.

$$
\begin{aligned}
&\exists S\, \forall k,t\ S(k,t) \to K(k) \wedge T(t) \wedge && (15)\\
&\quad \forall k \exists t\ S(k,t) \wedge && (16)\\
&\quad \forall k,t',t''\, S(k,t') \wedge S(k,t'') \to t' = t'' \wedge && (17)\\
&\quad \forall k',k'',j,t',t'',l'\ Job(k',j) \wedge Job(k'',j) \wedge \\
&\qquad k' \neq k'' \wedge Aft(k'',k') \wedge S(k',t') \wedge S(k'',t'') \wedge && (18)\\
&\qquad L(k',l')\ \to\ t'' \geq t' + l' \wedge \\
&\quad \forall k',k'',p,t',t'',l',l'' \\
&\qquad Proc(k',p) \wedge Proc(k'',p) \wedge k' \neq k'' \wedge L(k',l') \wedge \\
&\qquad L(k'',l'') \wedge S(k',t') \wedge S(k'',t'') \to && (19)\\
&\qquad\quad \big[(t' \geq t'' \to t' \geq t''+l'') \wedge (t' \leq t'' \to t'' \geq t'+l')\big] \wedge \\
&\quad \forall k,t,l\ \ T(k) \wedge S(k,t) \wedge L(k,l) \to Time(t+l). && (20)
\end{aligned}
$$

Constraints (15–17) 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, (17) *assigns at most one starting time to each task). Moreover, constraint* (18) *forces tasks that belong to the same job to be executed in their order without overlapping, while* (19) *avoids a processor to perform more than one task at each time point. Finally,* (20) *forces the scheduling to terminate before deadline D.*

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 (16));
- $\beta(k) \doteq \bigwedge_{t \neq \overline{t}} \neg S_t(k)$ (obtained by unfolding (17)).

α and β above clearly satisfy **Hyp 2** of Theorem 1. Moreover, according to the algorithm in Figure 5, by iteratively focusing on all predicates S_t, we can delay all such (unfolded) constraints. It is worth noting that the unfolding of guessed predicates is needed only to formally characterize the reformulation with respect to Theorem 1, and must not be performed in practice.

Intuitively, the constraint we delay, i.e. (17), imposes at most one start time for each task: 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 (Crawford & Baker 1994) 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.

Example 3. *In the Bin packing problem (Garey & Johnson 1979, Prob. SR1) (cf. also (Martello & Toth 1990)), 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:*

$$\begin{aligned} &\exists P\, \forall i, b\, P(i,b) \rightarrow I(i) \wedge B(b) \wedge && (21)\\ &\quad \forall i \exists b\, I(i) \rightarrow P(i,b) \wedge && (22)\\ &\quad \forall i, b, b'\, P(i,b) \wedge P(i,b') \rightarrow b = b' && (23)\\ &\quad \forall b, c\, C(b,c) \rightarrow \\ &\qquad\qquad sum\left(\{s \mid P(i,b) \wedge S(i,s)\}\right) \leq c && (24)\end{aligned}$$

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. 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 (21)*, total* (22)*, monodrome* (23)*, and satisfying the capacity constraint for every bin* (24)*.*

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 (23)*, that force an item to be packed in exactly one bin. Thus, by iteratively applying Theorem 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.*

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.

Non-shrink second stages. As specified in Section "Reformulation", 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.

Actually, there are other specifications which 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 4. *In the Golomb ruler problem (`www.csplib.org`, Prob. 3), 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: (i) mark i is put on the left (i.e., before) mark j if and only if $i < j$, and (ii) 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:*

$$\begin{aligned} &\exists G \forall m, i\, G(m,i) \rightarrow M(m) \wedge P(i) \wedge && (25)\\ &\quad \forall m \exists i\, M(m) \rightarrow P(m,i) \wedge && (26)\\ &\quad \forall m, i, i'\, G(m,i) \wedge G(m,i') \rightarrow i = i' \wedge && (27)\\ &\quad \forall m, m', i, i'\, G(m,i) \wedge G(m',i') \wedge m < m' \rightarrow i < i' \wedge && (28)\\ &\quad \forall m, m', i, i', n, n', j, j' \\ &\qquad G(m,i) \wedge G(m',i') \wedge G(n,j) \wedge G(n',j') \wedge && (29)\\ &\qquad m<m' \wedge n<n' \wedge (m<n \vee (m=n \wedge m'<n')) \rightarrow \\ &\qquad\quad (i'-i) \neq (j'-j). \end{aligned}$$

A solution is thus an extension for the guessed predicate G which is a mapping (25–27) assigning a point in the ruler to every mark, such that the order of marks is respected (28) *and distances between two different marks are all different* (29).

Differently from the previous examples, the constraint that can be delayed here is (28), that forces the ascending ordering among marks. By neglecting it, we extend the set of solutions of the original problem with all of 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 (28), 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. A similar kind of second stage is needed for reformulating some *permutation problems* by safe-delay.

Furthermore, a modification of some of the other constraints may be needed to ensure the correctness of the reformulation. As an example, in constraint (29) differences must be replaced by their absolute values.

We are currently investigating the formal aspects of such a generalization, and whether this kind of reformulations are effective in practice. Some preliminary results on reformulating a class of permutation problems that include, e.g., Hamiltonian path, Permutation flow-shop, and Tiling problems, appear in (Mancini 2003).

Methodological discussion

In this section we make a discussion on the methodology we adopted in this work, in particular the use of ESO as a modelling language, and the choice of the solvers for the preliminary experimentation.

Using ESO for specifying problems wipes out many aspects of state-of-the-art languages which are somehow difficult to take into account, thus simplifying the task of finding criteria for reformulating problem specifications. However, ESO, even if somewhat limited, is not too far away from the modelling languages provided by some commercial systems. An example is AMPL, which admits only linear constraints: in this case, the reformulation technique described in Theorem 1 can often be straightforwardly applied; for instance, a specification of the k-coloring problem in such a language is as follows:

```
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;
```

The reformulated specification can be obtained by simply replacing the "CoveringAndDisjointness" constraint with the following one:

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

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.

We also note that, as shown in Examples 2, 3, and 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 noted that a specification written in ESO naturally leads to a translation into a SAT instance. For this reason, we have chosen to use a SAT solver to start the experimentation of the proposed technique. Anyway, we repeated the experimentation using the state-of-the-art commercial systems CPLEX (linear) and SOLVER (non-linear). Thus, SAT-based experiments have to be considered as a starting point and an encouraging evidence of the reasonableness of the proposed approach, by showing that consistent speed-ups in the solving process can be obtained by a mere reformulation of a pure declarative specification. Actually, this evidence has been partially confirmed by the experiments done using the linear integer programming solver CPLEX: also in this case, several classes of instances benefit from safe-delay.

Experimental results

We made an experimentation of our reformulation techniques on 3-coloring (randomly generated instances), k-coloring (benchmark instances from the DIMACS repository `ftp://dimacs.rutgers.edu/pub/challenge`), and job-shop scheduling (benchmark instances from OR library `www.ms.ic.ac.uk/info.html`), solving each instance both with and without delaying the disjointness constraints, using both the DPLL-based SAT system SATZ (Li & Anbulagan 1997) (and an ad hoc program (Cadoli & Schaerf 2001) for the instantiation stage), and the state-of-the-art constraint and linear programming system OPL (Van Hentenryck 1999), obviously using it as a pure modelling language, and omitting search procedures. For what concerns the latter system, we wrote both a linear and a non-linear specification for the above problems, and applied our reformulation technique to the linear ones (cf. previous section). 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, both instantiation and post-processing times are negligible, and comparison can be done only on solving time.

Summing up, we solved several thousands of instances. For what concerns the SAT experimentation, it is worth noting that in all instances the SAT time without disjointness is less than or equal to the time with disjointness. As far as CPLEX is concerned, we found that several (but not all) instances benefitted from delaying constraints, especially those for which the linear specification is more efficient than the non-linear one (cf. the following paragraphs).

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

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

3-coloring. We solved the problem on 3,500 randomly generated graph instances with 430 nodes each. The number of edges varies, and covers the phase transition region (Cheeseman, Kanefski, & Taylor 1991): the ratio (# of directed edges/# of nodes) varies between 2 and 6. The average solving time (150 instances for each fixed number of edges) varies between fractions of a second and a minute. The saving percentage in the SAT-based experiments varies between 15% and 50%, for both positive and negative instances, the hardest instances being at 30%. Experimentation with CPLEX and SOLVER is planned.

k-coloring. Results of our SAT-based experiments are shown in Table 1 (n and e are the number of nodes and edges). The saving percentage varies between 9.0% and 59%. On the other hand, when applying the reformulation technique to OPL, we observed two major evidences:

1. It is not the case that a specification (linear or non-linear) is always more efficient than the other one. In particular, the ratio between the solving time of CPLEX and SOLVER is highly variable, and the linear specification can be much more efficient than the non-linear one, especially for *negative* instances.
2. By focusing on the class of instances for which the linear specification is more efficient than the non-linear one, we found out several instances in which delaying the disjointness constraint leads to appreciable time savings.

Job shop scheduling. We considered two instances known as FT06 (36 tasks, 6 jobs, 6 processors, solvable with deadline 55) and LA02 (50 tasks, 10 jobs, 5 processors, solvable with deadline 655). SAT solving times are listed in Table 2 for different values for the deadline. As it can be observed, the saving is quite consistent. For what concerns the experiments in OPL, CPLEX seems not to be affected much by delaying constraints (or even affected negatively), and anyway it is slower than SOLVER.

Summing up, delaying of constraints seems to be always effective when using a SAT solver. As far as CPLEX is concerned, we have mixed evidence. We plan to extend the experimentation to problems in which SAT performs worse than other solvers.

As concluding remarks, it has to be observed that, differently from the SAT-based experimentation, in the OPL-based one it not always the case that the reformulation technique leads to positive time savings. Consequently, further studies have to be done in order to better understand when and why the linear specification is more efficient than the non-linear one, and when and why delaying the disjointness constraint is profitable, when dealing with linear specifications. From our current point of view, it seems that the linear specification, at least for the above problems, is often more efficient when dealing with negative instances.

Conclusions, related and future work

In this paper 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, which often results in faster solving. In this way, we have shown that reasoning on a specification can be very effective.

Although Theorem 1 calls for a tautology checking (cf. **Hyp 2**), we have shown different specifications for which this test is easy. 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. As an example, in (Cadoli & Mancini 2004) we have shown that state-of-the-art ATPs usually perform very well in similar tasks (i.e., detecting and breaking symmetries and functionally dependent predicates on problem specifications).

Related work. Several researchers addressed the issue of reformulation of a problem *after the instantiation phase*: as an example, in (Weigel & Bliek 1998) it is shown how to translate an instantiated CSP into its Boolean form, which is useful for finding different reformulations, while in (Choueiry & Noubir 1998) the proposed approach is to generate a conjunctive decomposition of an instantiated CSP, by localizing independent subproblems. Finally, in (Freuder & Sabin 1997) it is shown that abstracting problems by simplifying constraints is useful for finding more efficient reformulations of the original problem; the abstraction may require backtracking for finding solutions of the original problem. In our work, we focus on reformulation of the specification, and, differently from other techniques, the approach is backtracking-free: once the first stage is completed, a solution will surely be found by evaluating the delayed constraints.

Other papers investigate the best way to encode an instance of a problem into a format adequate for a specific solver. As an example, many different ways for encoding graph coloring or permutation problems into SAT have been figured out, cf., e.g., (Frisch & Peugniez 2001; Walsh 2001). Conversely, we take a specification-oriented approach.

Finally, we point out that a logic-based approach has also been successfully adopted in the ‘80s to study the query optimization problem for relational DBs. Analogously to our approach, the query optimization problem has been attacked relying on the query (i.e., the specification) only, without considering the database (i.e., the instance), and it was firstly studied in a formal way using first-order logic (cf., e.g., (Aho & Ullman 1979; Chandra & Merlin 1977)). In a later stage, the theoretical framework has been translated into rules for the automated rewriting of queries expressed in real world languages and systems.

Future work. In this paper we have focused on a form of reformulation which 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., (12)), or to allow non-shrink second stages (cf., e.g., the specification for the Golomb ruler problem in Example 4), in order to allow reformulation for a larger class of specifications. 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. In fact, it not always the case that delaying constraints, and so making the set of solutions larger, improves the solving process. Adding, e.g., symmetry-breaking or implied constraints are well known techniques that may reach the same goal with the opposite strategy, i.e., reducing the set of solutions. Currently, it is not 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.

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.

Acknowledgements. This research has been supported by MIUR (Italian Ministry for Instruction, University, and Research) under the FIRB project ASTRO (Automazione dell'Ingegneria del Software basata su Conoscenza), and under 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".

Instance	Colors	Solvable?	SAT			CPLEX			SOLVER
			W/ disj.	W/o disj.	% saving	W/ disj.	W/o disj.	% saving	W/ disj.
le450_5d	5	Y	6.0	5.2	13.3	540.8	628.1	-16.1	1.2
le450_15a	13	N	–	–	–	8.6	6.3	26.7	–
le450_25a	21	N	–	–	–	83.6	17.2	79.4	–
DSJC125.9	21	N	–	–	–	1408.4	936.7	33.5	–
DSJC250.1	9	Y	1.6	0.9	43.8	–	–	–	12.5
DSJR500.1	11	N	–	–	–	2.2	1.3	40.9	297.4
DSJR500.1	12	Y	–	–	–	18.4	18.1	1.6	0.5
queen8_8	9	Y	1.5	1.2	20.0	–	2411.2	>33.0	1.9
queen9_9	10	Y	32.1	25.3	21.2	–	–	–	134.8
queen10_10	15	Y	1.3	0.9	30.8	3.3	3.1	6.1	0.9
queen11_11	13	Y	22.9	17.7	22.7	–	125.1	>96.5	41.7
queen12_12	15	Y	1.3	0.8	38.5	463.7	228.3	50.8	9.7
fpsol2.i.2	21	N	–	–	–	21.8	13.6	37.6	–

(‘–’ means that the solver did not terminate in one hour)

Table 1: Solving times (seconds) for k-coloring.

Instance	Deadline	Solvable?	SAT			CPLEX			SOLVER
			W/ disj.	W/o disj.	% saving	W/ disj.	W/o disj.	% saving	W/ disj.
FT06	100	Y	–	4.6	$\sim$100	117.4	–	$\sim -\infty$	1.1
FT06	65	Y	3.7	1.7	52.8	–	–	–	1.2
FT06	55	Y	4.0	1.6	60.7	–	–	–	1.5
FT06	54	N	16.1	4.3	73.6	–	–	–	–
FT06	52	N	2.0	1.0	48.0	–	–	–	7.2
LA02	1200	Y	4.6	2.1	53.9	31.2	–	$\sim -\infty$	1.5
LA02	1000	Y	2.9	1.5	48.8	–	–	–	1.6
LA02	960	Y	22.2	1.8	91.8	–	–	–	–
LA02	860	Y	740.0	8.5	98.8	–	–	–	–
LA02	840	Y	–	15.1	$\sim$100	–	–	–	–

(‘–’ means that the solver did not terminate in one hour)

Table 2: Solving times (seconds) for job shop scheduling.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison Wesley Publ. Co., Reading, Massachussetts.

Aho, A.V., S. Y., and Ullman, J. 1979. Equivalences among relational expressions. *SIAM Journal on Computing* 2(8):218–246.

Börger, E.; Gräedel, E.; and Gurevich, Y. 1997. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer.

Cadoli, M., and Mancini, T. 2004. Using a theorem prover for reasoning on constraint problems. In *Proceedings of the International Workshop on Constraint Programming and Constraints for Verification (CP+CV'04), at the European Joint Conferences on Theory and Practice of Software (ETAPS'04).*

Cadoli, M., and Schaerf, A. 2001. Compiling problem specifications into SAT. In *Proceedings of the European Symposium On Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, 387–401. Springer.

Chandra, A. K., and Merlin, P. M. 1977. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the Ninth ACM Symposium on Theory of Computing (STOC'77)*, 77–90. Boulder, CO, USA.

Cheeseman, P.; Kanefski, B.; and Taylor, W. M. 1991. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, 163–169.

Choueiry, B. Y., and Noubir, G. 1998. On the computation of local interchangeability in discrete constraint satisfaction problems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, 326–333.

Crawford, J. M., and Baker, A. B. 1994. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, 1092–1097.

Dechter, R. 1992. *Constraint Networks (Survey).* John Wiley & Sons, Inc. 276–285.

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. The KR system `dlv`: Progress report, Comparisons and Benchmarks. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, 406–417.

Fagin, R. 1974. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In Karp, R. M., ed., *Complexity of Computation*, 43–74. AMS.

Fourer, R.; Gay, D. M.; and Kernigham, B. W. 1993. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson Publishing.

Freuder, E. C., and Sabin, D. 1997. Interchangeability supports abstraction and reformulation for multi-dimensional constraint satisfaction. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, 191–196.

Frisch, A. M., and Peugniez, T. J. 2001. Solving non-boolean satisfiability problems with stochastic local search. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, 282–290.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability—A guide to NP-completeness*. San Francisco: W.H. Freeman and Company.

Li, C., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, 366–371.

Mancini, T. 2003. Reformulation techniques for a class of permutation problems. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP'03)*, number 2833 in Lecture Notes in Computer Science, 984.

Martello, S., and Toth, P. 1990. *Knapsack Problems: algorithms and computer implementation*. John Wiley & Sons Ltd.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3,4):241–273.

Van Hentenryck, P. 1999. *The OPL Optimization Programming Language*. The MIT Press.

Walsh, T. 2001. Permutation problems and channelling constraints. In *Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'01)*, number 2250 in Lecture Notes in Computer Science, 377–391. Springer.

Weigel, R., and Bliek, C. 1998. On reformulation of constraint satisfaction problems. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI'98)*, 254–258.