Towards automated reformulation of specifications

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. Several state-of-the-art systems and languages for constraint solving adopt a clear separation between the *specification* of a problem and its *instances*. Some of them additionally perform a limited form of *reasoning* on the spec, with the goal of choosing the most appropriate solver. In this paper we propose a more sophisticated form of reasoning on problem specs, with the goal of *reformulating* them so that they are more efficiently solvable. To this end, we present a reformulation technique that highlights constraints that can be safely "delayed", and solved afterwards. Our main contribution is the characterization (with soundness proof) of safe-delay constraints wrt a syntactic criterion on the spec, thus obtaining a mechanism for the automated reformulation of specs applicable to a great variety of problems, e.g., graph coloring and job shop scheduling. Another contribution is a preliminary experimentation on the effectiveness of the proposed technique, which reveals promising time savings.

1 Introduction

Several systems and languages for the solution of constraint problems adopt a clear separation between the *specification* of a problem, e.g., graph three-coloring, and its *instance*, e.g., *a* graph. On top of that, they use a two-level architecture for finding solutions: the specification is *instantiated* against the instance, and then an appropriate solver is invoked. Examples of systems of such kind are AMPL [11], OPL [20], GAMS [5], DLV [9], SMODELS [18], and NP-SPEC [4].

The major benefit of this separation 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, e.g., AMPL, are able to translate –at the request of the user– a spec in various formats, suitable for different solvers.

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.

In this paper we propose a more sophisticated architecture, in which reasoning on the specification is more complex. Our goal is to *reformulate* the

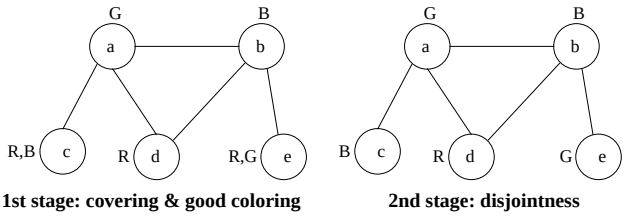


Fig. 1. Delaying the disjointness constraint in 3-coloring

specification so that the later stages of computation, i.e., instantiation and solution, are more efficient. 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., [1]).

Reformulation is quite difficult 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 [3]. For this reason we limit our attention on restricted forms of reformulation, and, more specifically, we focus on a reformulation that selects 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. Fig. 1). We call a constraint with this property a *safe-delay 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 offers several advantages:

- The instantiation phase will be faster, since safe-delay constraints are not taken into account.
- Solving the simplified problem, i.e., the one without disjointness, might be easier than the original formulation. 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.

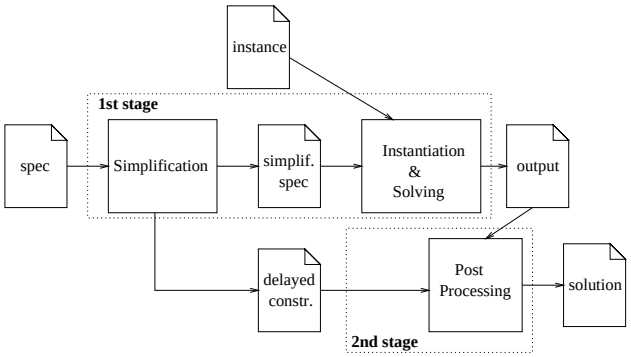


Fig. 2. Reformulation architecture

– Ad hoc efficient methods for solving delayed constraints may exist. In fact, 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 Fig. 2 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 syntactic criterion on the specification. This allows us to obtain a mechanism for the automated reformulation of a specification (shown in Sec. 3) that can be applied to a great variety of problems, including the so-called *functional* ones.

Another contribution is an experimentation on the effectiveness of the proposed reformulation, shown in Sec. 4, on both benchmark and randomly generated instances. Finally, a discussion on the adopted methodology is given in Sec. 5, while conclusions and related work are described in Sec. 6.

Other researchers have dealt with delaying constraints on *instantiated* problems, as an example, CLP($\mathcal{R}$) delays evaluation of non-linear constraints. What characterizes our contribution (cf. Sec. 6) is a study of safe-delaying performed on specifications.

2 Preliminaries

For the specification of problems, in this paper we use *existential second-order logic* (ESO), which allows to represent all search problems in the complexity class NP [10]. The use of ESO as a modelling language for problem specs is common in the database literature, but unusual in constraint programming,

therefore few comments are in order. Constraint modelling systems like those mentioned in Sec. 1 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 specs written in higher-level languages. In Sec. 5 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.

In the following, σ denotes a fixed set of relational symbols not including equality "=", and $S_1, \ldots, S_h$ denote variables ranging over relational symbols distinct from those in $\sigma \cup \{=\}$. By Fagin's theorem [10], any collection $\mathbf{D}$ of finite databases over σ is recognizable in NP time iff it is defined by an ESO formula of the kind:

$$\exists S_1, \ldots, S_h \ \phi, \tag{1}$$

where $S_1, \ldots, S_h$ are relational variables of various arities and ϕ is a function-free first-order formula containing occurrences of relational symbols from $\sigma \cup \{S_1, \ldots, S_h\} \cup \{=\}$. The symbol "=" is always interpreted in the obvious way, i.e., as "identity".

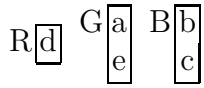
A database D is in $\mathbf{D}$ iff there is a list of relations $\Sigma_1, \ldots, \Sigma_h$ (matching the list of relational variables $S_1, \ldots, S_h$) which, along with D, satisfies formula (1), i.e., such that $(D, \Sigma_1, \ldots, \Sigma_h) \models \phi$. The tuples of $\Sigma_1, \ldots, \Sigma_h$ must take elements from the *Herbrand universe* of D, i.e., the set of constant symbols occurring in it.

Example 1. In the "three-coloring" NP-complete decision problem (cf. [14, Prob. GT4, p. 191]) 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*:

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

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

Referring to the graph in Fig. 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. Fig. 1, right):



In what follows, existentially quantified predicates (like R, G, and B) will be called *guessed*, and the set of tuples from the Herbrand universe they take will be called their *extension* and denoted with $ext()$. As an example, $ext(G) = \{a, e\}$ in the previous example. 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.

3 Reformulation

In this section we show sufficient conditions for constraints of a specification to be safe-delay. We refer to the architecture of Fig. 2, with some general assumptions:

1. As shown in Fig. 1, 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. Fig. 3 represents the extensions of the red predicate in the first (R^*) and second (R) stages of Fig. 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.

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.

3.1 Single monadic predicate

We refer to the 3-coloring specification of Example 1, focusing on the guessed predicate R, and trying to find an intuitive explanation for the fact that clauses

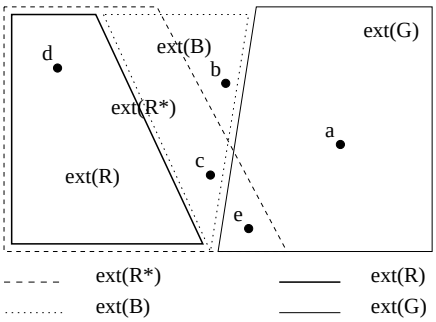


Fig. 3. Extensions for the 3-coloring spec

(3-4) can be delayed. We immediately note that clauses in the specification can be partitioned into three subsets:

NO: R does not occur in them, i.e., (5,7,8);
NEG: R occurs negatively in them, i.e., (3,4,6);
POS: R occurs positively in them, i.e., (2).

Neither **NO** nor **NEG** 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** clauses are not safe-forget –because shrinking the extension of R can violate them– we now show that clause (2) it is. In fact, if we equivalently rewrite clauses (2) and (3-4), respectively, as follows:

$$\forall X \; \neg B(X) \wedge \neg G(X) \rightarrow R(X) \qquad (2)'$$
$$\forall X \quad R(X) \rightarrow \neg B(X) \wedge \neg G(X), \qquad (3-4)'$$

we note that clause $(2)'$ sets a lower bound for the extension of R, and clauses $(3\text{-}4)'$ 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 $ext(R^*)$, 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. Fig. 3). Next theorem shows that is not by chance that the antecedent of $(2)'$ is semantically related to the consequence of $(3\text{-}4)'$.

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

$$\exists S_1, \ldots, S_h, R \quad \Xi \quad \wedge \quad \forall X \; \alpha(X) \rightarrow R(X) \quad \wedge \quad \forall X \; R(X) \rightarrow \beta(X), \qquad (9)$$

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 Φ^s be:

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

where R^* is a new predicate symbol, and Ξ^* is Ξ with R replaced by R^*. Let Φ^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 $(D, M^s) \models \Phi^s$, then $(D, M^s - ext(R^*), ext(R)) \models \Phi$, where $ext(R)$ is the extension of R as defined by M^s and Φ^d.

Referring to Fig. 2, Φ is the spec, D is the instance, Φ^s is the "simplified spec", and $\forall X\ R(X) \rightarrow \beta(X)$ is the "delayed constraint". Solving Φ^s against D produces –if the instance is satisfiable– a list of extensions M^s (the "output"). Evaluating Φ^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 (thus in polynomial time).

In other words, the theorem says that, for each satisfiable instance D of the simplified specification Φ^s, each solution M^s of Φ^s can be translated, via Φ^d, to a solution of the original specification Φ; 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)$'$. Fig. 3 represents possible extensions of the red predicate in the first (R^*) and second (R) stages, for the instance of Fig. 1.

Proofs are omitted for lack of space. As evidence for the soundness of Theorem 1, we claim that Fig. 4 (left) shows that, if **Hyp 2** holds, then 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 Φ by the following proposition.

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

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

$$\forall X\ edge(X, X) \rightarrow R(X), \tag{10}$$

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

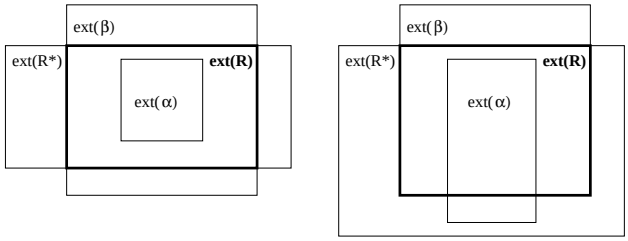


Fig. 4. Extensions with and without **Hyp 2**

either because of (2) or because of (10), 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 (10), **Hyp 1** is clearly not satisfied. Analogously, if (10) 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. Fig. 4, right, gives further evidence that the constraint $\forall X \ \alpha(X) \rightarrow R(X)$ can be violated if $ext(R)$ is computed using Φ^d and $ext(\alpha)$ is not a subset of $ext(\beta)$.

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 s.t. $M \models \Xi$, shrinking $ext(R)$ and keeping everything 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*.

3.2 Set of monadic predicates

Theorem 1 can be applied recursively to the specification Φ^s, by focusing on a different guessed predicate, in order to obtain a new simplified specification $(\Phi^s)^s$ and new delayed constraints $(\Phi^s)^d$. Since, by Proposition 1, satisfiability of such formulae is preserved, it is afterwards possible to translate, via $(\Phi^s)^d$, each solution of $(\Phi^s)^s$ to a solution of Φ^s, and then, via Φ^d, to a solution of Φ.

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

Algorithm SOLVEBYDELAYING
Input a specification Φ, 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(Φ);

```
   if (⟨Φ̄ˢ, D⟩ is satisfiable) then
   begin
      let M be a solution of ⟨Φ̄ˢ, D⟩;
      while (Φ̄ᵈ is not empty) do
      begin
         Constraint d = Φ̄ᵈ.pop();
         M = M∪ solution of d; // cf. Theorem 1
      end;
      return M;
   end;
   else return 'unsatisfiable';
end;

Procedure Reformulate
Input a specification Φ
Output ⟨Φ̄ˢ, Φ̄ᵈ⟩, where Φ̄ˢ is a simplified spec, and Φ̄ᵈ is a stack of delayed constraints
begin
   Stack Φ̄ᵈ = the empty stack;
   Φ̄ˢ = Φ;
   for each monadic guessed predicate R in Φ̄ˢ do
   begin
      partition constraints in Φ̄ˢ according to Theorem 1, in
      ⟨Ξ;   ∀X α(X) → R(X);   ∀X R(X) → β(X)⟩;
      if (previous step is possible with ∀X β(X) ≠ TRUE) then
      begin
         Φ̄ᵈ.push('∀X R(X) ↔ R*(X) ∧ β(X)');
         Φ̄ˢ = Ξ* ∧ ∀X α(X) → R*(X);
      end;
   end;
   return ⟨Φ̄ˢ, Φ̄ᵈ⟩;
end;
```

As an example, we evaluate the procedure Reformulate on the spec of Example 1, by focusing on the guessed predicates in the order R, G, B. The output is the following simplified specification $\overline{\Phi^s}$:

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

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

$$\forall X \;\; R(X) \leftrightarrow R^*(X) \wedge \neg G(X) \wedge \neg B(X); \tag{11}$$

$$\forall X \;\; G(X) \leftrightarrow G^*(X) \wedge \neg B(X). \tag{12}$$

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 $\overline{\Phi^d}$.

From any solution of $\overline{\Phi^s}$, a solution of Φ is obtained in the **while** loop of SolveByDelaying by reconstructing first of all the extension for G by formula (12), and then the extension for R by formula (11) (synthesized, respectively, in the second and first iteration of the algorithm). Since each constraint

in the stack $\overline{\Phi^d}$ is first-order, the whole **while** loop 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.

3.3 Binary predicates

The specification of the k-coloring problem using a binary predicate Col – the first argument being the node and the second the color– is as follows (constraints represent, respectively, covering, disjointness, and good coloring).

$$\begin{aligned} &\exists Col\, \forall X\ \exists Y\ Col(X,Y) \wedge \\ &\qquad \forall XYZ\ Col(X,Y) \wedge Col(X,Z) \rightarrow Y = Z \wedge \\ &\qquad \forall XYZ\ X \neq Y \wedge Col(X,Z) \wedge Col(Y,Z) \rightarrow \neg edge(X,Y). \end{aligned} \tag{13}$$

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 spec of Example 1.

The above considerations imply that we can use the architecture of Fig. 2 for a large class of specifications.

Definition 1. *A* safe-delay functional *spec is an ESO formula of the form:*

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

where Ξ is a conjunction of clauses in which P either does not occur or occurs negatively.

The term "functional" originates from covering and disjointness constraints, which imply a search space which is a (total) function from a finite domain to a finite codomain. 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 and algorithm SOLVEBYDELAYING.

Safe-delay functional specs are quite common; apart from graph coloring, a notable example (formal details omitted) is Job shop scheduling [14, Prob. SS18]: there are n jobs, m tasks, and p processors. Jobs are ordered collections of tasks and each task has an integer-valued length and the processor that performs it. Each processor can perform a task at the time, and the 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. The disjointness constraint imposes at most one starting time for each task: thus, by applying Theorem 1 for delaying it, we allow a task to have multiple starting times, all of them being good ones. In the PostProcessing stage we can arbitrarily choose one of them to obtain a solution of the original problem.

4 Experimental results

We made a preliminary experimentation of our reformulation techniques on 3-coloring (randomly generated instances), k-coloring (instances taken from the DIMACS benchmark repository `ftp://dimacs.rutgers.edu/pub/challenge`), and job shop scheduling (benchmark instances taken from the OR library at `www.ms.ic.ac.uk/info.html`). According to the results of Sec. 3, we solved each instance both with and without delaying the disjointness constraints. The specifications were instantiated against the instances using ad hoc programs, thus obtaining SAT instances. As for the solver, we used the DPLL-based SAT system SATZ, described in [17].

Few methodological comments are mandatory. It is well-known that the benchmark problems we refer to can be efficiently solved either by means of ad hoc algorithms or by using constraint programming languages in a sophisticated way. Our purpose here is not to propose an efficient method for problem solving, but rather to prove that we can achieve a consistent speed-up by means of a mere reformulation of a purely declarative spec, even if in a language –ESO– with no built-in constructs for functions or integers. So, the experimentation should be considered only as a preliminary stage to test whether the reformulation approach we propose is promising or not.

Experiments were executed on a SparcStation Ultra2 and on a Pentium 4. 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 30 minutes. In this way, both instantiation and postprocessing, i.e., evaluation of delayed constraints, times are negligible, and comparison can be done only on SAT time.

Summing up, we solved several thousands of instances. It is worth noting that in all of them the SAT time without disjointness is less than or equal to the time with disjointness. In what follows, we refer to the *saving percentage*, defined as the ratio:

$$(time\_with\_disjointness - time\_without\_disjointness)/time\_with\_disjointness.$$

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 [7]: 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 210 seconds. The saving percentage varies between 15% and 50%, the hardest instances being at 30%.

k-coloring Our results are shown in Table 1 (n is the number of nodes and e of the edges). The saving percentage varies between 9.6% and 55%.

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

Table 1. Solving times for k-coloring

Graph	Nodes	Edges	Colors	SAT time (secs)		
				With disj.	Without disj.	% saving
DSJC250.1	250	6436	9	8.16	4.99	38.9
le450_5a	450	5714	5	492.93	428.78	13.0
le450_5b	450	5734	5	434.05	378.45	12.8
le450_5c	450	9803	5	22.45	20.02	10.8
le450_5d	450	9757	5	29.78	26.93	9.6
miles500	128	2340	20	8.36	3.76	55.0
queen9_9	81	2112	10	175.54	137.45	21.7
queen10_10	100	2940	20	7.00	4.16	40.6
queen12_12	144	5192	15	8.25	6.12	25.8

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 ('–' means that SAT with disjointness constraints did not terminate in 21 hours).

Table 2. Solving times for job shop scheduling

Instance	Deadline	Solvable?	SAT time (secs)		
			With disj.	Without disj.	% saving
FT06	55	Y	7.80	3.07	60.6
FT06	54	N	29.23	7.59	74.0
LA02	960	Y	40.13	3.39	91.5
LA02	860	Y	1308.63	15.60	98.8
LA02	840	Y	–	76.36	∼100

5 Methodological discussion

In this section we make a discussion on the methodology we adopted in this work: the use of ESO as a modelling language, and the use of a SAT solver for the experimentation.

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 syntactic criteria for reformulating problem specs. As mentioned before, we plan to study richer languages (mentioned in Sec. 1) in a further stage.

However, it must be observed that ESO, even if somewhat limited, is not too far away from the modelling languages provided by some commercial systems. An example of such a language 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 reasonable spec of the k-coloring problem in such a language is the following:

```
param n_nodes;   param n_colors integer, > 0;
set NODES := 1..n_nodes;   set EDGES within NODES cross NODES;
set COLORS := 1..n_colors;
var Coloring {NODES,COLORS} binary;   # Coloring as a binary predicate
s.t. CoveringAndDisjointness {x in NODES}:
    sum {c in COLORS} Coloring[x,c] = 1;   # nodes have exactly one color
s.t. GoodColoring {(x,y) in EDGES, c in COLORS}:
    Coloring[x,c] + Coloring[y,c] <= 1;   # nodes linked by an edge have diff. colors
```

The above spec is similar to (13), and the reformulated spec can be obtained by rewriting the "CoveringAndDisjointness" constraint in the following way:

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

For what concerns the experimentation stage, it is worthwhile to mention that a spec written in ESO naturally leads to a translation into a SAT instance. In this sense, the use of a SAT solver should not be considered as the ultimate goal of our research: state-of-the-art systems and languages usually perform much better on the kind of problems of Sec. 4 (although other problems, e.g., planning ones [15], can be efficiently solved by SAT). Actually, it has to be considered only as a starting point of our research, and an encouraging evidence of the reasonableness of the proposed approach. A more complex experimentation using state-of-the-art systems as those listed in Sec. 1 will be conducted soon to understand whether the technique discussed so far and its variants are applicable in the new contexts. Linear solvers like CPLEX (which, e.g., AMPL can use) will be particularly useful, due to the considerations made above.

6 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 results in faster solving. In this way, we have shown that reasoning on a spec can be very effective.

Related work Several researchers addressed the issue of reformulation of a problem *after the instantiation phase*. As an example, in [12] it is shown that abstracting problems by simplifying constraints is useful for finding more efficient reformulations of the original problem; abstraction may require backtracking for finding solutions of the original problem. In our work, we focus on reformulation of the spec, and the approach is backtracking-free: once the first stage is completed, a solution will surely be found by evaluating the delayed constraints.

Analogously, in [22] it is shown how to translate an instantiated CSP into its boolean form, which is useful for finding different reformulations. Finally, in [8] it is shown how to generate a conjunctive decomposition of an instantiated CSP, by localizing independent subproblems.

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., [13, 21]. Conversely, in our work 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 the approach presented in this paper, the query optimization problem has been attacked relying on the query (i.e., the spec) 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., [2, 6, 16, 19]). 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.

Future work In this paper we have focused on a form of reformulation which partitions the first-order part of a spec. This basic idea can be generalized, as an example by evaluating in both stages of the computation a constraint (as an example (10)), in order to allow reformulation for a larger class of specs. 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.

Moreover, we wish to extend our results to the so-called *permutation* problems, i.e., problems which, on top of the constraints of Definition 1 have also the following constraints:

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

However, this kind of specs do not belong to the class studied in this paper, and a generalization of Theorem 1 is needed to support them.

Permutation problems arise frequently in theory and practice: examples are the *n-queens*, the *Hamiltonian circuit*, and the *code generation for parallel assignments* (resp. [14, Prob. GT39, PO6]) problems.

Finally, it is our goal to move the above described and forecoming theoretical results into rules for automatically reformulate problem specs given in much more complex languages, e.g. AMPL and OPL, which have higher-level built-in constructs, such as integers and functions.

References

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

2. S. Y. Aho, A.V. and J. Ullman. Equivalences among relational expressions. *SlAM Journal on Computing*, 2(8):218–246, 1979.
3. E. Börger, E. Gräedel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.
4. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. In *Proceedings of the European Symposium On Programming (ESOP 2001)*, volume 2028 of *LNCS*, pages 387–401. Springer, 2001.
5. E. Castillo, A. J. Conejo, P. Pedregal, and N. A. Ricardo Garca. *Building and Solving Mathematical Programming Models in Engineering and Science*. Wiley, 2001.
6. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th ACM Symp. Theory of Computing (STOC'77)*, pages 77–90. Boulder, CO, USA, 1977.
7. P. Cheeseman, B. Kanefski, and W. M. Taylor. Where the really hard problem are. In *Proc. of IJCAI'91*, pages 163–169, 1991.
8. B. Y. Choueiry and G. Noubir. On the computation of local interchangeability in discrete constraint satisfaction problems. In *Proc. of AAAI'98*, pages 326–333, 1998.
9. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, Comparisons and Benchmarks. In *Proc. of KR'98*, pages 406–417, 1998.
10. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In R. M. Karp, editor, *Complexity of Computation*, pages 43–74. AMS, 1974.
11. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson Publishing, 1993.
12. E. C. Freuder and D. Sabin. Interchangeability supports abstraction and reformulation for multi-dimensional constraint satisfaction. In *Proc. of AAAI'97*, pages 191–196, 1997.
13. A. M. Frisch and T. J. Peugniez. Solving non-boolean satisfiability problems with stochastic local search. In *Proc. of IJCAI 2001*, pages 282–290, 2001.
14. M. R. Garey and D. S. Johnson. *Computers and Intractability—A guide to NP-completeness*. W.H. Freeman and Company, San Francisco, 1979.
15. H. A. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proc. of KR'96*, pages 374–384, 1996.
16. A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 1(35):146–160, 1988.
17. C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proc. of IJCAI'97*, pages 366–371, 1997.
18. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
19. Y. Sagiv and M. Yannakakis. Equivalence among relational expressions with the union and difference operations. *Journal of the ACM*, 4(27):633–655, 1980.
20. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
21. T. Walsh. Permutation problems and channelling constraints. In *Proc. of LPAR 2001*, number 2250 in LNCS, pages 377–391. Springer, 2001.
22. R. Weigel and C. Bliek. On reformulation of constraint satisfaction problems. In *Proc. of ECAI'98*, pages 254–258, 1998.