Available online at www.sciencedirect.com
SCIENCE DIRECT®
Artificial Intelligence 170 (2006) 779–801
Artificial Intelligence
www.elsevier.com/locate/artint

Automated reformulation of specifications by safe delay of constraints ☆

Marco Cadoli, Toni Mancini *

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, I-00198 Roma, Italy

Received 14 September 2004; received in revised form 25 January 2006; accepted 25 January 2006

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, bin-packing, 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 an experimentation on the effectiveness of the proposed technique using six different solvers, which reveals promising time savings.

Keywords: Modelling; Reformulation; Second-order logic; Propositional satisfiability; Constraint satisfaction problems

1. Introduction

Current state-of-the-art languages and systems for constraint modelling and programming (e.g., AMPL [22], OPL [48], XPRESS^{MP},[1] GAMS [9], DLV [31], SMODELS [39], ESRA [21], PS [18] and NP-SPEC [8]) exhibit a strong separation between a problem *specification* (e.g., Graph 3-coloring) and its *instance* (e.g., a graph), usually adopting a two-level architecture for finding solutions: the specification is firstly instantiated (or grounded) against the instance, and then an appropriate solver is invoked (cf. Fig. 1). Such a separation leads to several advantages: obviously declarativeness increases, and the solver is completely decoupled from the specification. Ideally, the programmer can focus only on the combinatorial aspects of the problem specification, without committing *a priori* to a specific solver. In

☆ This paper is an extended and revised version of [M. Cadoli, T. Mancini, Automated reformulation of specifications by safe delay of constraints, in: Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2004), Whistler, BC, Canada, AAAI Press/The MIT Press 2004, pp. 388–398].

* Corresponding author.
E-mail addresses: cadoli@dis.uniroma1.it (M. Cadoli), tmancini@dis.uniroma1.it (T. Mancini).

[1] Cf. http://www.dashoptimization.com.

0004-3702/$ – see front matter
doi:10.1016/j.artint.2006.01.008

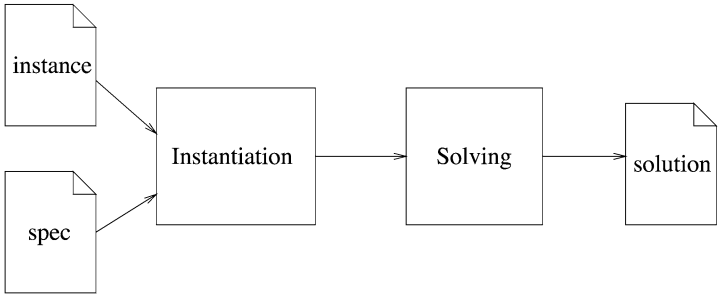

Fig. 1. Two-level architecture of current problem solving systems.

fact, some systems, e.g., AMPL, are able to translate—at the request of the user—a specification in various formats, suitable for different solvers, e.g., among the others, CPLEX, MINOS,[2] LANCELOT.[3]

Nonetheless, many existing techniques proposed in the literature for optimizing the solution of constraint satisfaction problems apply after the commitment to the instance: notable examples are, e.g., symmetry detection and breaking (cf., e.g., [4,16,37]), the development of techniques for imposing various local consistency notions and of heuristics during search (cf., e.g., [17]), the development of algorithms that deal with dependent variables, e.g., those added to SAT instances during the clausification of non-CNF formulae [27], and with the so-called "equivalence clauses" [32].

However, in many cases, properties that are amenable to be optimized derive from the problem structure, rather than the particular instance considered. Optimization techniques that act on the problem structure have been proposed. They include the addition of implied constraints (cf., e.g., [47]), the deletion or abstraction of some of the constraints (cf., e.g., [28]), the use of redundant models, i.e., multiple viewpoints synchronized by channelling constraints, in order to increase constraint propagation [12,20,29].

Our research follows the latter approach, with the aim of systematize the process of finding useful reformulations by performing a *symbolic reasoning* on the specification. In general, for many properties, symbolic reasoning can be more natural and effective than making such "structural" aspects emerge after instantiation, when the structure of the problem has been hidden.

An example of system that performs a sort of reasoning on the specification is OPL, which is able to automatically choose the most appropriate solver for a problem. However, the kind of reasoning offered is very primitive: OPL only checks (syntactically) whether a specification is linear, in this case invoking a linear—typically more efficient—solver, otherwise a general constraint programming one.

Conversely, our research aims to the following long-term goal: the *automated reformulation* of a declarative constraint problem specification, into a form that is more efficiently evaluable by the solver at hand. The ultimate goal is to handle all properties suitable for optimization that derive from the problem structure at the specification level, leaving at the subsequent instance level the handling of the remaining ones, i.e., those that truly depend on the instance. In fact, it is worthwhile to note that focusing on the specification does not rule out the possibility of additionally applying existing optimization techniques at the instance level.

The approach we follow is similar, in a sense, to the one used in the database research community for attacking the query optimization problem in relational databases. A query planner, whose task is to reformulate the query posed by the user in order to improve the efficiency of the evaluation, takes into account the query and the database schema only, not its current content, i.e., the instance (cf., e.g., [1]).

In general, reformulating a constraint problem specification is a difficult task: 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, research must focus on controlled and restricted forms of reformulation.

Moreover, the effectiveness of a particular reformulation technique is expected to depend both on the problem and on the solver, even if it is possible, in principle, to find reformulations that are good for all solvers (or for solvers of a certain class, e.g., linear, or SAT-based ones). To this end, in related work (cf. Section 6), we present different reformulation strategies that have been proposed in order to speed-up the process of solving a constraint problem.

[2] Cf. http://www.sbsi-sol-optimize.com/.

[3] Cf. http://www.cse.clrc.ac.uk/nag/lancelot/lancelot.shtml.

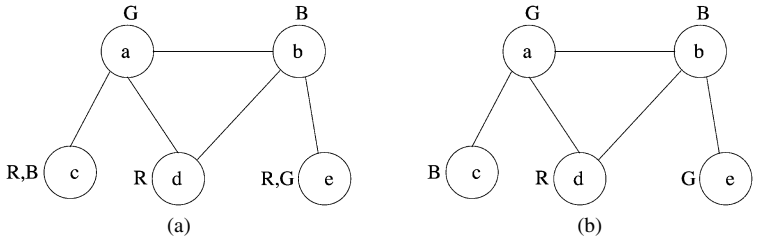

Fig. 2. Delaying the disjointness constraint in 3-coloring. (a) 1st stage: covering and good coloring; (b) 2nd stage: disjointness.

In this paper, we propose a technique that allows us to select constraints in a problem specification that can be ignored in a first step (regardless of the instance), and efficiently reinforced once a solution of the simplified problem has been found. We call such constraints *safe-delay*. Moreover, we experimentally show how reformulating problem specifications by safe-delay improves performances of different (but not all) solvers. On one hand, this gives evidence that problem reformulation can be effective in many cases, and on the other, it confirms the intuition that a single reformulation technique may have positive effects for some classes of solvers, but negative ones for others, and that a portfolio of different and complementary reformulation strategies has to be considered, in general (cf. Section 6 for related work).

The NP-complete graph k-coloring problem offers a simple example of a safe-delay constraint. 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 at a later stage (cf. Fig. 2). It is interesting to note that the deletion of the disjointness constraints in graph k-coloring has been already proposed as an ad-hoc technique in [46] (cf. also [42]), and implemented in, e.g., the standard DIMACS formulation in SAT of k-coloring.

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. Fig. 1) will typically be faster, since safe-delay constraints are not taken into account. As an example, let's assume we want to use (after instantiation) a SAT solver for the solution of k-coloring on a graph with n nodes and e edges. The SAT instance encoding the k-coloring instance—in the obvious way, cf., e.g., [25]—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 need not to be generated.
- Solving the simplified problem, i.e., the one without disjointness, might be easier than the original formulation for some classes of solvers, since removing constraints makes the set of solutions larger. For each instance it holds that:

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

  In our experiments, using six different solvers, including SAT, integer linear programming, and constraint programming ones, we obtained fairly consistent (in some cases, more than one order of magnitude) speed-ups for hard instances of various problems, e.g., graph coloring and job-shop scheduling. On top of that, we implicitly obtain several good solutions. Results of the experimentation are given in Section 5.
- 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 Fig. 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, cf., e.g., [14], but

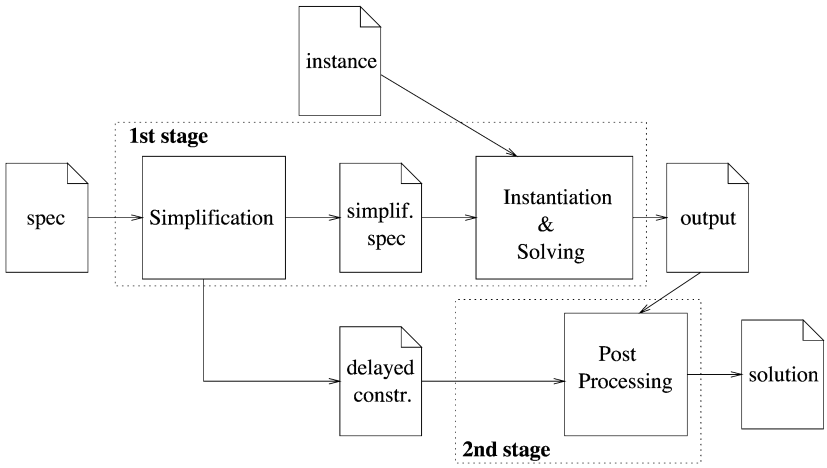

Fig. 3. Reformulation architecture.

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, i.e., those in which the search space is a (total) function from a finite domain to a finite codomain.

The outline of the paper is as follows: after recalling some preliminaries in Section 2, we present our reformulation technique in Section 3, and a discussion on the adopted methodology in Section 4. Afterwards, experimentation on the effectiveness of the approach is described in Section 5, on both benchmark and randomly generated instances, using SAT and state-of-the-art linear and constraint programming solvers. Finally, conclusions, future and related work are presented in Section 6.

2. Preliminaries

The style used for the specification of a combinatorial problem varies a lot among different languages for constraint programming. In this paper, rather than considering procedural encodings such as those obtained using libraries (in, e.g., C++ or PROLOG), we focus on highly declarative languages. Again, the syntax varies a lot among such languages: AMPL, OPL, XPRESS^{MP} and GAMS allow the representation of constraints by using algebraic expressions, while DLV, SMODELS, and NP-SPEC are rule-based languages. Anyway, from an abstract point of view, all such languages are extensions of *existential second-order logic* (ESO) over finite databases, where the existential second-order quantifiers and the first-order formula represent, respectively, the *guess* and *check* phases of the constraint modelling paradigm. In particular, in all such languages it is possible to embed ESO queries, and the other way around is also possible, as long as only finite domains are considered.

To this end, in this paper we use ESO for the specification of problems, mainly because of its simplicity and because it allows to represent all search problems in the complexity class NP [19,40]. In particular, as we show in the remainder of this section, ESO can be considered as the formal basis for virtually all available languages for constraint modelling. Intuitively, the relationship between ESO and real modelling languages is similar to that holding between Turing machines or assembler, and high-level programming languages. 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 4 we discuss further our choice, showing how our reformulation technique can be easily lifted in order to deal with problem specifications written in other and richer languages, e.g., AMPL.

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), with no function symbols.

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 $\pi(\mathcal{I})$ and extensions of predicates in $\vec{S}$ which verify $\phi(\vec{S}, \mathcal{I})$. More formally, the following must hold:

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

It is worthwhile to note that, when a specification is instantiated against an input database, a constraint satisfaction problem (in the sense of [17]) is obtained.

Example 1. (*Graph* 3-*Coloring* [26, Prob. GT4]) In this NP-complete decision problem 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*:

$$\exists RGB \quad \forall X \quad R(X) \vee G(X) \vee B(X) \ \wedge \tag{2}$$

$$\forall X \quad R(X) \rightarrow \neg G(X) \ \wedge \tag{3}$$

$$\forall X \quad R(X) \rightarrow \neg B(X) \ \wedge \tag{4}$$

$$\forall X \quad B(X) \rightarrow \neg G(X) \ \wedge \tag{5}$$

$$\forall XY \ X \neq Y \wedge R(X) \wedge R(Y) \rightarrow \neg edge(X, Y) \wedge \tag{6}$$

$$\forall XY \ X \neq Y \wedge G(X) \wedge G(Y) \rightarrow \neg edge(X, Y) \wedge \tag{7}$$

$$\forall XY \ X \neq Y \wedge B(X) \wedge B(Y) \rightarrow \neg edge(X, Y), \tag{8}$$

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

Finally, we observe that in this paper we consider basic ESO. Nonetheless, it is known (cf., e.g., [35]) that much syntactic sugar can be added to ESO in order to handle types, functions, bounded integers and arithmetics, without altering its expressing power. In Section 3.3 we give some examples of the enriched language.

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. 3, with some general assumptions:

Assumption 1. As shown in Fig. 2, the output of the first stage of computation may—implicitly—contain *several* solutions. As an example, node c can be assigned to either green or blue, and node e to either red or green. In the second stage we do not want to compute all of them, but just to arbitrarily select one. In other words, we focus on search problems, with no objective function to be optimized.

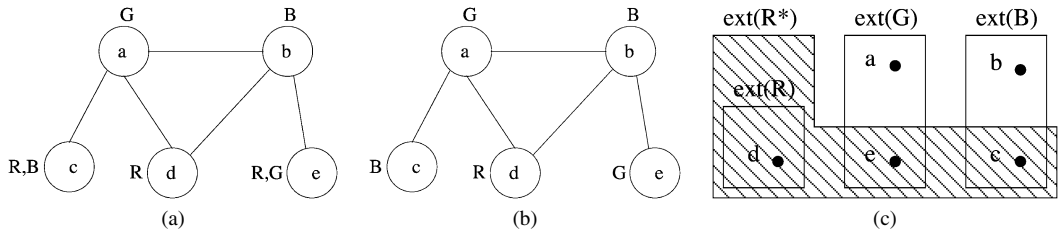

Fig. 4. (a) A solution of the first (a) and second (b) stage of a graph 3-coloring instance. (c) An alternative view showing how the extension for R shrinks when moving from the first (dashed area) to the second stage.

Assumption 2. The second stage of computation can only *shrink* the extension of a guessed predicate. Fig. 4 represents a solution of the first (a) and second (b) stage on the graph 3-coloring instance in Fig. 2. Fig. 4(c) gives further evidence about how the extension for predicate R shrinks when moving from the first ($ext(R^*)$) to the second stage ($ext(R)$) ($ext(B)$ and $ext(G)$ are unchanged).

This assumption is coherent with the way most algorithms for constraint satisfaction operate: each variable has an associated *finite domain*, from which values are progressively eliminated, until a satisfying assignment is found. Nonetheless, in Section 3.4 we give examples of problem specifications which are amenable to safe-delay, although with a second stage of different nature.

Identification of safe-delay constraints requires reasoning on the whole specification, taking into account relations between guessed and database predicates. For the sake of simplicity, in Section 3.1 we will initially focus our attention on *a single monadic* guessed predicate, trying to figure out which constraints concerning it can be delayed. Afterwards, in Section 3.2 we extend our results to *sets* of monadic guessed predicates, then, in Section 3.3, to *binary* predicates.

3.1. 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, and POS_R with—respectively—no, only negative, and only positive occurrences of R.

Neither NO_R nor NEG_R clauses can be violated by shrinking the extension of R. Such constraints will be called *safe-forget* for R, because if we decide to process (and satisfy) them in the first stage, they can be safely ignored in the second one (which, by Assumption 2 above, can only shrink the extension for R). We note that this is just a possibility, and we are not obliged to do that: as an example, clauses (3–4) will *not* be evaluated 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 safe-forget. In fact, if we equivalently rewrite clauses (2) and (3–4), respectively, as follows:

$$\forall X \quad \neg B(X) \wedge \neg G(X) \rightarrow R(X) \tag{2}'$$

$$\forall X \quad R(X) \rightarrow \neg B(X) \wedge \neg G(X), \tag{3–4}'$$

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. Fig. 4). The next theorem (all proofs are delayed to Appendix A) 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*:

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

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

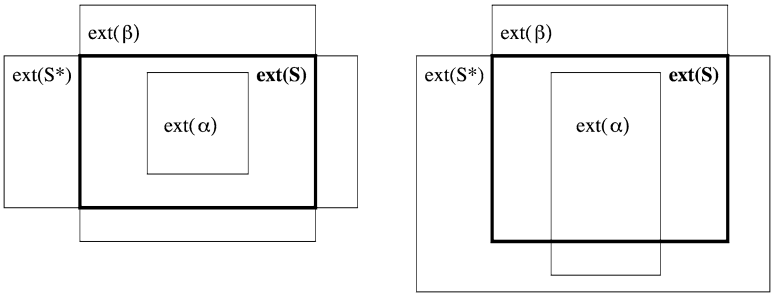

Fig. 5. Extensions with and without Hyp 2.

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

Let now ψ^s *be*:

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

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

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

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

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

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

A comment on the relevance of the above theorem is in order. Referring to Fig. 3, ψ is the specification, $\mathcal{I}$ is the instance, ψ^s is the "simplified specification", and $\forall X\ S(X) \rightarrow \beta(X)$ is the "delayed constraint" (belonging to NEG_S). Solving ψ^s against $\mathcal{I}$ produces—if the instance is satisfiable—a list of extensions M^* (the "output"). Evaluating ψ^d against M^* corresponds to the "PostProcessing" phase in the second stage. The structure of the delayed constraint ψ^d clearly reflects Assumption 2 above, i.e., that extensions for guessed predicates can only be shrunk in the second stage. Moreover, since the last stage amounts to the evaluation of a first-order formula against a fixed database, it can be done in logarithmic space (cf., e.g., [1]), thus in polynomial time.

In other words, Theorem 1 says that, for each satisfiable instance $\mathcal{I}$ of the simplified specification ψ^s, each solution M^* of ψ^s can be translated, via ψ^d, to a solution of the original specification ψ; we can also say that $\Xi \wedge \forall X\ \alpha(X) \rightarrow S(X)$ is safe-forget, and $\forall X\ S(X) \rightarrow \beta(X)$ is safe-delay.

Referring to the specification of Example 1, the distinguished guessed predicate is R, 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. 4 represents possible extensions of the red predicate in the first (R^*) and second (R) stages, for the instance of Fig. 2, and Fig. 5 (left) gives further evidence that, if Hyp 2 holds, the constraint $\forall X\ \alpha(X) \rightarrow S(X)$ can never be violated in the second stage.

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

Theorem 2. *Let* $\mathcal{I}$*,* ψ*,* ψ^s *and* ψ^d *as in Theorem* 1*. For every instance* $\mathcal{I}$*, if* ψ *is satisfiable,* ψ^s *and* ψ^d *are satisfiable.*

To substantiate the reasonableness of the two hypotheses of Theorem 1, we play the devil's advocate and consider the following example.

Example 2. (*Graph* 3*-Coloring with red self-loops* (*Example* 1 *continued*)) In this problem, which is a variation of the one in Example 1, the input is the same as for graph 3-Coloring, and the question is whether it is possible to find a coloring of the graph with the additional constraint that all self-loops insist on red nodes.

A specification for this problem can be easily derived from that of Example 1 by adding the following constraint:

$$\forall X\ edge(X, X) \rightarrow R(X). \tag{9}$$

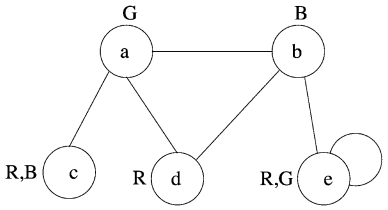

Fig. 6. An instance of the graph 3-coloring with red self-loops problem along with a solution of the first stage, obtained by delaying only the disjointness constraints.

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 (9), 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 (9), Hyp 1 is clearly not satisfied. Analogously, if (9) 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. 5, right, gives further evidence that the constraint $\forall X\ \alpha(X) \rightarrow S(X)$ can be violated if $ext(S)$ is computed using ψ^d and $ext(\alpha)$ is not a subset of $ext(\beta)$. An instance of the graph 3-coloring with red self-loops problem is given in Fig. 6, along with a solution of the first stage, in the case Ξ contains also the constraint (9). It can be observed that constraints (3–4) are not enough to set the correct color for node e.

Summing up, a constraint with a positive occurrence of the distinguished guessed predicate S can be safely forgotten (after being evaluated in the first stage) only if there is a safe-delay constraint which justifies it.

Some further comments about Theorem 1 are in order. As it can be observed (cf. Appendix A), the theorem proof does not formally require Ξ to be a conjunction of clauses; actually, it can be any formula such that, from any structure M such that $M \models \Xi$, by shrinking $ext(S)$ and keeping everything else fixed we obtain another model of Ξ. As an example, Ξ may contain the conjunct $\exists X\ S(X) \rightarrow \gamma(X)$ (with $\gamma(X)$ a first-order formula in which S does not occur). Secondly, although Hyp 2 calls for a tautology check—which is not decidable in general—we will see in what follows that many specifications satisfy it *by design*.

3.2. Set of monadic predicates

Theorem 1 states that we can delay some constraints of a specification ψ, by focusing on one of its monadic guessed predicates, hence obtaining a new specification ψ^s, and a set of delayed constraints ψ^d. Of course, the same theorem can be further applied to the specification ψ^s, by focusing on a different guessed predicate, in order to obtain a new simplified specification $(\psi^s)^s$ and new delayed constraints $(\psi^s)^d$. Since, by Theorem 2, 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 Fig. 7 deals with the general case of a set of guessed predicates: if the input specification ψ is satisfiable, it returns a simplified specification $\overline{\psi^s}$ and a list of delayed constraints $\overline{\psi^d}$. Algorithm SOLVEBYDELAYING gets any solution of $\overline{\psi^s}$ and translates it, via the evaluation of formulae in the list $\overline{\psi^d}$—with LIFO policy—to a solution of ψ.

As an example, by evaluating the procedure REFORMULATE on the specification of Example 1, by focusing on the guessed predicates in the order R, G, B, we obtain as output the following simplified specification $\overline{\psi^s}$, that omits all disjointness constraints (i.e., clauses (3–5)):

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

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

$$\forall X\ R(X) \leftrightarrow R^*(X) \wedge \neg G(X) \wedge \neg B(X); \tag{10}$$

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

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 ($\langle \overline{\Phi^s}, D\rangle$ 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 Φ;
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. S in $\overline{\Phi^s}$ **do**
    begin
      partition constraints in $\overline{\Phi^s}$ according to Thm 1, in:
        $\langle \Xi;\quad \forall X\ \alpha(X) \rightarrow S(X);\quad \forall X\ S(X) \rightarrow \beta(X)\rangle$;
      if the previous step is possible with $\forall X\ \beta(X) \neq$ TRUE **then**
        begin
          $\overline{\Phi^d}$**.push**('$\forall X\ S(X) \leftrightarrow S^*(X) \wedge \beta(X)$');
          $\overline{\Phi^s} = \Xi^* \wedge \forall X\ \alpha(X) \rightarrow S^*(X)$;
        end;
    end;
  return $\langle \overline{\Phi^s}, \overline{\Phi^d}\rangle$;
end;

Fig. 7. Algorithm for safe-delay in case of a set of monadic predicates.

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

From any solution of $\overline{\psi^s}$, a solution of ψ is obtained by reconstructing first of all the extension for G by formula (11), and then the extension for R by formula (10) (synthesized, respectively, in the second and first iteration of the algorithm). Since each delayed constraint is first-order, the whole second stage is doable in logarithmic space (thus in polynomial time) in the size of the instance.

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

3.3. Binary predicates

In this subsection we show how our reformulation technique can be extended in order to deal with specifications with binary (and, in general, n-ary) guessed predicates. This can be formally done by unfolding non-monadic guessed predicates into monadic ones, exploiting the finiteness of the Herbrand domain.

To illustrate the point, 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 (the input schema in this case is given by

$\vec{R} = \{node(\cdot), color(\cdot), edge(\cdot,\cdot)\}$ encoding the set of nodes, colors, and the graph edges, respectively, and constraints force *Col* to be correctly typed, and to satisfy conditions for covering, disjointness, and good coloring):

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

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 replace the binary predicate *Col* with three monadic guessed predicates Col_1, Col_2, Col_3, one for each value of the second argument (i.e., the color), with the meaning that, if tuple $\langle n, c\rangle$ belongs to *Col*, then $\langle n\rangle$ belongs to Col_c. Constraints of the specification must be unfolded accordingly. The output of the unfolding process for $k = 3$—up to an appropriate renaming of Col_1, Col_2, Col_3 into R, G, B—is exactly the specification of Example 1.

The above considerations imply that we can use the architecture of Fig. 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

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

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

Example 3. (*Job-shop scheduling* [26, Prob. SS18]) In the Job-shop scheduling problem we have sets (sorts) J for jobs, K for tasks, and P for processors. Jobs are ordered collections of tasks and each task has an integer-valued length (encoded in binary relation L) and the processor that is requested in order to perform it (in binary relation *Proc*). Each processor can perform a task at the time, and tasks belonging to the same job must be performed in their order. Finally, there is a global deadline D that has to be met by all jobs.

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

$$\exists S\ \ \forall k,t\ S(k,t) \rightarrow K(k) \wedge T(t)\ \wedge \tag{12}$$

$$\forall k\ \exists t\ S(k,t)\ \wedge \tag{13}$$

$$\forall k,t',t''\ S(k,t') \wedge S(k,t'') \rightarrow t' = t''\ \wedge \tag{14}$$

$$\begin{aligned}&\forall k',k'',j,t',t'',l'\ Job(k',j) \wedge Job(k'',j) \wedge \\ &\quad k' \neq k'' \wedge Aft(k'',k') \wedge S(k',t') \wedge S(k'',t'') \wedge \\ &\quad L(k',l')\ \rightarrow\ t'' \geqslant t' + l'\ \wedge\end{aligned} \tag{15}$$

$$\begin{aligned}&\forall k',k'',p,t',t'',l',l'' \\ &\quad Proc(k',p) \wedge Proc(k'',p) \wedge k' \neq k'' \wedge L(k',l') \wedge \\ &\quad L(k'',l'') \wedge S(k',t') \wedge S(k'',t'') \rightarrow \\ &\quad\ \big[(t' \geqslant t'' \rightarrow t' \geqslant t'' + l'') \wedge (t' \leqslant t'' \rightarrow t'' \geqslant t' + l')\big] \wedge\end{aligned} \tag{16}$$

$$\forall k,t,l\ \ T(k) \wedge S(k,t) \wedge L(k,l) \rightarrow Time(t+l). \tag{17}$$

Constraints (12–14) 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, (14) assigns at most one starting time to each task). Moreover, constraint (15) forces tasks that belong to the same job to be executed in their order without overlapping, while (16) avoids a processor to perform more than one task at each time point. Finally, (17) forces the scheduling to terminate before deadline D.

It is worth noting that additional syntactic sugar may be added to ESO in order to better deal with scheduling problems. As an example, constructs like those commonly found in richer modelling languages for such problems (cf., e.g., the part of OPL concerning scheduling) can be made available. However, such enhancements are out of the scope of this paper, and will not be taken into account. □

As an example, Fig. 8(a) and (b) show, respectively, an instance and a possible solution of the Job-shop scheduling problem. The instance consists of 3 jobs (J1, J2, and J3) of, respectively, 4, 3, and 5 tasks each. The order in which tasks belonging to the same job have to be performed is given by the letter in parentheses (a, b, c, d, e). Tasks have to be executed on 3 processors, P1, P2, and P3, which are denoted by different borderlines. Hence, the processor needed to perform a given task is given by the task borderline.

To reformulate the Job-shop scheduling problem, after unfolding the specification in such a way to have one monadic guessed predicate S_t for each time point t, we focus on a time point $\bar{t}$ and partition clauses in the specification in which $S_{\bar{t}}$ 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:

- $\alpha(k) \doteq \bigwedge_{t \neq \bar{t}} \neg S_t(k)$ (obtained by unfolding (13));
- $\beta(k) \doteq \bigwedge_{t \neq \bar{t}} \neg S_t(k)$ (obtained by unfolding (14)).

α and β above clearly satisfy Hyp 2 of Theorem 1. Moreover, according to the algorithm in Fig. 7, 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. (14), forces each task to have at most one starting time: thus, by delaying it, we allow a task to have multiple starting times, i.e., the task does not overlap with any other task at any of its start times. Again, in the second stage, we can arbitrarily choose one of them. We observe that a similar approach has been used in [15] 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. As an example, Fig. 8(c) shows a solution of the first stage of the reformulated problem for the instance in Fig. 8(a), which subsumes the solution in Fig. 8(b).

Example 4. (*Bin packing* [26, Prob. SR1]) In the Bin packing problem (cf. also [36]), we are asked to pack a set I of items, each one having a given size, into a set B of bins, each one having a given capacity. Under the assumption that input instances are given as extensions for relations I, S, B, and C, where I encodes the set of items, B the set of bins, S the size of items (a tuple $\langle i, s\rangle$ for each item i), and C the capacity of bins (a tuple $\langle b, c\rangle$ for each bin b), an ESO specification for this problem is as follows:

$$\exists P \quad \forall i, b\ P(i,b) \rightarrow I(i) \wedge B(b)\ \wedge \tag{18}$$

$$\forall i\ \exists b\ I(i) \rightarrow P(i,b)\ \wedge \tag{19}$$

$$\forall i, b, b'\ P(i,b) \wedge P(i,b') \rightarrow b = b' \tag{20}$$

$$\forall b, c\ C(b,c) \rightarrow sum\big(\{s \mid P(i,b) \wedge S(i,s)\}\big) \leqslant c \tag{21}$$

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

In the above specification, a solution is a total mapping P from items to bins. Constraints force the mapping to be, respectively, over the right relations (18), total (19), mono-valued (20), and satisfying the capacity constraint for every bin (21).

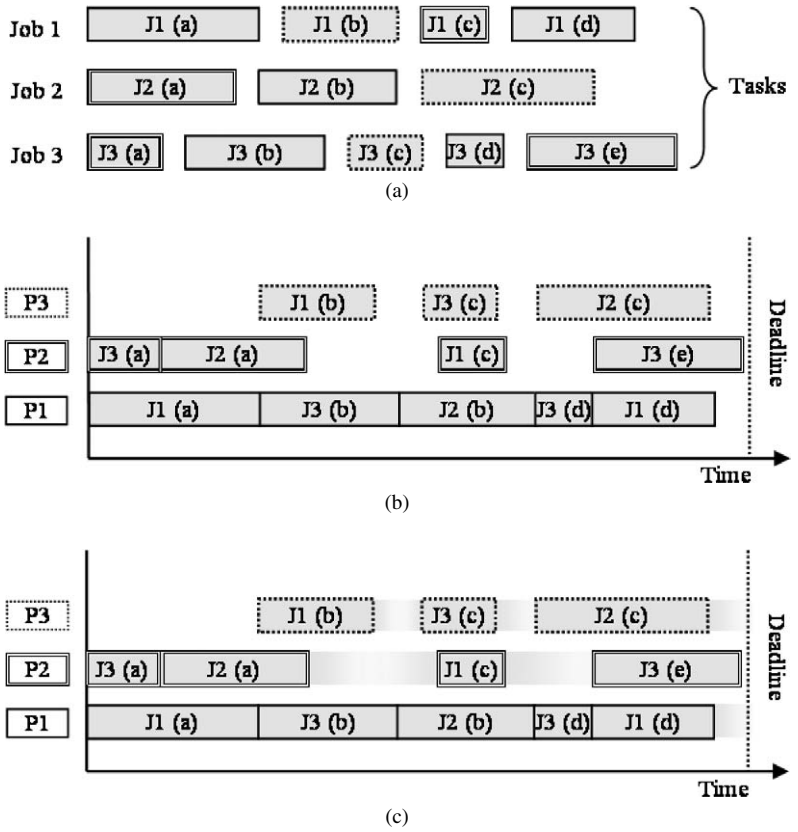

Fig. 8. (a) An instance of the Job-shop scheduling problem, consisting in 3 jobs of, respectively, 4, 3, and 5 tasks each, to be executed on 3 processors. (b) A possible solution of the whole problem for the instance in (a). (c) A solution of the first stage of the reformulated problem, that subsumes that in (b). Shades indicate multiple good starting times for tasks.

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 (20), 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.

Other problems that can be reformulated by safe-delay exist. Some examples are Schur's Lemma (www.csplib.org, Prob. 15) and Ramsey problem (www.csplib.org, Prob. 17). A short discussion on how these reformulations can be addressed is given in [7].

As showed in the previous examples, it is worth noting that arithmetic constraints do not interfere with our reformulation technique. As an instance, in the last example, the "$\leqslant$" predicate leads to clauses that remain satisfied if the extension of the selected guessed predicate is shrunk, while keeping everything else fixed.

3.4. Non-shrink second stages

As specified at the beginning of Section 3, in this paper 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 5. (*Golomb ruler* (www.csplib.org, Prob. 6)) In this problem, we are asked to put m marks $M_1, \ldots, M_m$ on different points on a ruler of length l in such a way that:

(1) Mark i is put on the left (i.e., before) mark j if and only if $i < j$, and
(2) The $m(m-1)/2$ distances among pairs of distinct marks are all different.

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

$$\exists G \quad \forall m, i\ G(m,i) \rightarrow M(m) \wedge P(i)\ \wedge \tag{22}$$

$$\forall m\ \exists i\ M(m) \rightarrow P(m,i)\ \wedge \tag{23}$$

$$\forall m, i, i'\ G(m,i) \wedge G(m,i') \rightarrow i = i'\ \wedge \tag{24}$$

$$\forall m, m', i, i'\ G(m,i) \wedge G(m',i') \wedge m < m' \rightarrow i < i' \wedge \tag{25}$$

$$\forall m, m', i, i', n, n', j, j'$$
$$G(m,i) \wedge G(m',i') \wedge G(n,j) \wedge G(n',j') \wedge \tag{26}$$
$$m < m' \wedge n < n' \wedge (m < n \vee (m = n \wedge m' < n')) \rightarrow$$
$$(i' - i) \neq (j' - j).$$

A solution is thus an extension for the guessed predicate G which is a mapping (22–24) assigning a point in the ruler to every mark, such that the order of marks is respected (25) and distances between two different marks are all different (26).

Here, the constraint that can be delayed is (25), which forces the ascending ordering among marks. By neglecting it, we extend the set of solutions of the original problem with all their permutations. In the second stage, the correct ordering among marks can be enforced in polynomial time.

By unfolding the binary guessed predicate G, we obtain $|M|$ monadic predicates G_m, one for each mark m. Once a solution of the simplified specification has been computed, by focusing on all of them, in order to reinforce the $m(m-1)/2$ unfolded constraints derived from (25), we possibly have to *exchange* tuples among pairs of predicates G_m and $G_{m'}$, for all $m \neq m'$, and not to shrink the extensions of single guessed predicates. Hence, Theorem 1 does not apply. Furthermore, a modification of some of the other constraints may be needed to ensure the correctness of the reformulation. In particular, in constraint (26) differences must be replaced by their absolute values.

As for the effectiveness of such a reformulation, it can be objected that the constraints delayed actually break the permutation symmetry, and removing them is likely to be not a good choice. However, even if this is likely to be true for CP solvers, like those based on backtracking, it is not obvious for others, e.g., SAT ones. In [7] we show how delaying such constraints significantly drops down the instantiation time needed by the NP-SPEC SAT compiler, without major variations in the solving times of the best SAT solver (ZCHAFF).

Another class of problems that are amenable to be reformulated by safe-delay is an important subclass of permutation problems, that includes, e.g., Hamiltonian path (HP), Permutation flow-shop, and Tiling. Some preliminary results on how these problems can be reformulated appear in [34]. As an example, HP can be reformulated by looking for (small) cliques in the graph, and viewing them as single nodes. If we find an HP of the reduced graph, we can, in polynomial time, obtain a valid solution of the original problem, since cliques can be traversed in any order.

We are currently investigating the formal aspects of such a generalization, and for which class of solvers this kind of reformulations are effective in practice.

4. 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 experimentation.

As already claimed in Section 1, and as the previous examples show, using ESO for specifying problems wipes out many aspects of state-of-the-art languages which are somehow difficult to take into account (e.g., numbers, arithmetics, constructs for functions, etc.), thus simplifying the task of finding criteria for reformulating problem specifications. However, it must be observed that ESO, even if somewhat limited, is not too far away from the modelling languages provided by some commercial systems.

A good example of such a language is AMPL [22], which admits only linear constraints: in this case, the reformulation technique described in Theorem 1 can often be straightforwardly applied; as an 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;
```

thus leading exactly to the reformulated specification of Example 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. Of course, study of safe-delay constraints is relevant also for such languages, because we can always specify in OPL problems such as the one of Example 5, which has such constraints, or we may want to write a linear specification for a given problem, in order to use a linear solver, more efficient in many cases (cf. Section 5).

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

As already claimed in Section 1, the effectiveness of a reformulation technique is expected to strongly depend on the particular solver used. To this end, we solved the same set of instances with SAT solvers of very different nature (cf. Section 5). Finally, since it is well-known that state-of-the-art linear and constraint programming systems may perform better than SAT on some problems, we repeated the experimentation by using commercial systems CPLEX (linear) and SOLVER (non-linear), invoked by OPLSTUDIO.[4]

5. Experimental results

We made an experimentation of our reformulation technique on 3-coloring (randomly generated instances), k-coloring (benchmark instances from the DIMACS repository[5]), and job-shop scheduling (benchmark instances from OR library[6]), using both SAT-based solvers (and the NP-SPEC SAT compiler [8] for the instantiation stage), and the constraint and linear programming system OPL [48], obviously using it as a pure modelling language, and omitting search procedures.

[4] Cf. http://www.ilog.com.

[5] Cf. ftp://dimacs.rutgers.edu/pub/challenge.

[6] Cf. http://www.ms.ic.ac.uk/info.html.

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

Und. edges	e/n	ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
		$\sum$ No delay	$\sum$ Delay	% sav.	$\sum$ No delay	$\sum$ Delay	% sav.	$\sum$ No delay	$\sum$ Delay	% sav.	$\sum$ No delay	$\sum$ Delay	% sav.
500	2.0	0.26	0.26	0.00	0.92	0.60	34.55	2.43	1.85	23.97	10.17	7.17	29.50
600	2.4	0.13	0.22	−69.23	2.22	1.82	18.05	3.63	3.00	17.43	8.97	5.08	43.37
700	2.8	0.22	0.12	45.45	8.48	7.58	10.61	9.92	8.60	13.28	7210.96	10804.02	−49.83
800	3.2	0.2	0.1	50.00	9.33	8.58	8.04	10.95	9.82	10.35	7208.19	7240.53	−0.45
900	3.6	0.18	0.14	22.22	18.18	16.52	9.17	19.88	17.87	10.14	14408.3	14403.53	0.03
1000	4.0	0.16	0.07	56.25	48.77	42.82	12.20	48.62	42.97	11.62	18787.4	21603.45	−14.99
1100	4.4	11.25	12.07	−7.29	119.70	105.50	11.86	119.47	105.47	11.72	11825.44	8772.81	25.81
1200	4.8	80537.81	86843.41	−7.83	129.35	112.05	13.37	129.75	113.78	12.31	16686.21	12279.19	26.41
1300	5.2	1497.69	1441.24	3.77	134.50	115.80	13.90	138.37	117.58	15.02	700.14	544.35	22.25
1400	5.6	60.4	59.77	1.04	145.32	119.93	17.47	145.53	124.68	14.33	95.09	78.27	17.69
1500	6.0	18.77	24.1	−28.40	147.68	125.25	15.19	152.82	128.97	15.61	31.56	26.95	14.61

As for the SAT-based experimentation, we used four solvers of very different nature: the DPLL-based complete systems ZCHAFF [38] and SATZ [33], and the local-search based incomplete solvers WALKSAT [45] and BG-WALKSAT [51] (the last one being guided by "backbones"). We solved all instances both with and without delaying constraints. As for OPL, we wrote both a linear and a non-linear specification for the above problems, and applied our reformulation technique to the linear one (cf. Section 4). All solvers have been used with their default parameters, without any heuristic or tuning that would possibly alter their performances. This is coherent with the declarative approach we adopted in this paper.

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 a few seconds, and less than one hour. In this way, both instantiation and post-processing, i.e., evaluation of delayed constraints, times are negligible, and comparison can be done only on the solving time.

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

$$(time\_no\_delay - time\_delay)/time\_no\_delay$$

3-*coloring*. We solved the problem on 1500 randomly generated graph instances with 500 nodes each. The number of edges varies, and covers the phase transition region [11]: the ratio (# of directed edges/# of nodes) varies between 2.0 and 6.0. In particular, we considered sets of 100 instances for each fixed number of edges, and solved each set both with and without delaying disjointness constraints (timeout was set at 1 hour). Table 1 shows overall solving times for each set of instances, for all the SAT solvers under consideration.

As it can be observed, the saving percentage depends both on the edges/nodes ratio, and on the solver. However, we have consistent time savings for many classes of instances. In particular, ZCHAFF seems not to be positively affected by safe-delay, and, for some classes of instances, e.g., those with 1500 edges, it seems to be negatively affected (−28.40%). On the other hand, both local-search based SAT solvers, i.e., WALKSAT and BG-WALKSAT show a consistent improvement with our reformulation technique, saving between 13% and 17% for hard instances. This behavior is consistent with the observation that enlarging the set of solutions can be profitable for this kind of solvers, and will be discussed at the end of this section, since it has been observed in all our experiments. Even if these are incomplete solvers, they have been always able to find a solution for satisfiable instances, except for the class of input graphs with 1100 edges: in this case, they found a solution on the 40% (WALKSAT) and 50% (BG-WALKSAT) of positive instances. Delaying disjointness constraints does not alter this percentage in a significant way. Also SATZ benefits from safe-delay, with savings between 22% and 26% for hard instances, even if underconstrained instances (e.g., those with 700 edges) highlight poorer performances (−49.83%).

It is worth noting, from the experiments described above, that the effectiveness of the reformulation may depend also on some instance-dependent parameters like, e.g., the edges/nodes ratio. However, this does exclude that some classes of solvers often benefit from the technique. In particular, it is interesting to note that the best technology, i.e., local search, is always improved.

For what concerns CPLEX instead, experimental results do not highlight significant variations in performances, since, in many cases, for the same set of 1500 instances, either both solving times were negligible, or a timeout occurred both with and without disjointness constraints. Finally, by solving the linear specification with SOLVER we observed the following mixed evidence: (i) About 15% of the instances were not solved, regardless of safe-delay; (ii) Another 15% of the instances were successfully solved with the original specification, but performing safe-delay on them prevented the system from terminate within the time limit; (iii) As for the remaining instances, average savings in time were often appreciable.

k-coloring. We solved the k-coloring problem on several benchmark instances of various classes of the DIMACS repository, with k close to the optimum, in order to have non-trivial instances, both positive and negative.

Results of our SAT-based experiments are shown in Table 2. As it can be observed, also here the effectiveness of the reformulation technique varies among the different solvers. In particular, ZCHAFF benefits by safe-delay on several instances, both positive and negative, but not on all of them. On the other hand, for local search solvers WALKSAT and BG-WALKSAT, delaying disjointness constraints always (except for very few cases) speeds-up the computation (usually by 20–30%). The same happens when using SATZ with even higher savings, even if this solver timeouts for several instances.

As for OPL instead, we have mixed evidence, since it is not the case that the linear specification (solved using CPLEX) is always more efficient than the non-linear one (solved using SOLVER), or vice versa. Indeed, the linear specification, when solved with CPLEX, often, but not always, benefits from safe-delay. Table 3 shows results obtained on graphs of various classes of the benchmark set. Differently from Table 2, in this case, due to the higher number of instances solved, we opted for showing aggregate results, grouping together instances of the same class. In partic-

Table 2
Solving times (seconds) for k-coloring (SAT solvers)

Instance	Colors	Solvable?	ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
			No delay	Delay	% sav.	No delay	Delay	% sav.	No delay	Delay	% sav.	No delay	Delay	% sav.
anna	10	N	24.87	15.02	39.61	6.48	5.71	11.83	6.2	4.0	4.57	–	–	–
anna	11	Y	0.01	0.01	0.00	0.08	0.05	40.00	0.1	0.06	33.33	0.29	0.09	68.97
david	10	N	15.15	13.04	13.93	5.58	4.78	14.33	5.52	4.66	15.41	–	–	–
david	11	Y	0.1	0.1	0.00	0.07	0.05	25.00	0.07	0.05	25.00	0.63	0.08	87.30
DSJC125.5	8	N	41.42	4.04	90.25	mem	mem	–	mem	mem	–	–	–	–
DSJC250.5	10	N	–	52.69	>98.54	mem	mem	–	mem	mem	–	–	–	–
DSJC500.1	5	N	11.14	1.55	86.09	mem	mem	–	mem	mem	–	158.67	145.33	8.41
le450_5a	5	Y	17.23	0.91	94.72	5.42	4.91	9.23	5.51	4.97*	9.97	96.83	77.16	20.31
le450_5b	5	Y	27.77	1.36	95.10	6.17*	5.10	17.30	6.15*	5.48	10.84	104.62	93.33	10.79
le450_5c	5	Y	0.02	0.02	0.00	10.00*	8.67	13.33	9.65	9.23*	4.32	22.45	20.02	10.82
le450_5c	9	Y	11.20	1.81	83.84	1.20	1.20	0.00	1.47	1.35	7.95	–	–	–
le450_5d	5	Y	0.09	1.20	−1233.33	8.52	8.15	4.31	8.42	8.40	0.20	6.00	5.20	13.33
miles500	9	N	80.19	54.55	31.97	9.08	8.51	6.24	9.70	7.38	23.92	–	–	–
miles500	20	Y	0.01	0.01	0.00	0.53	0.36	31.25	0.63	0.46	26.32	8.36	3.76	55.02
mulsol.i.2	30	N	–	–	–	28.83	22.85	20.75	30.18	24.16	19.93	–	–	–
mulsol.i.2	31	Y	0.01	0.01	0.00	4.25	2.36	44.31	4.21	2.80	33.60	5.10	2.67	47.65
mulsol.i.3	30	N	–	–	–	29.20	23.03	21.12	31.47	24.38	22.51	–	–	–
mulsol.i.3	31	Y	0.01	0.01	0.00	4.22	2.48	41.11	4.60	3.05	33.70	5.04	2.70	46.43
mulsol.i.4	30	N	–	–	–	29.40	23.58	19.78	30.93	24.81	19.77	–	–	–
mulsol.i.4	31	Y	0.01	0.01	0.00	4.03	2.38	40.91	4.98	2.76	44.48	5.08	2.73	46.26
mulsol.i.5	30	N	–	–	–	26.70	21.38	19.91	28.48	22.63	20.54	–	–	–
mulsol.i.5	31	Y	0.01	0.01	0.00	4.63	2.53	45.32	5.42	2.88	46.77	5.14	2.74	46.69
myciel5	5	N	413.99	1714.26	<−314.08	1.33	1.16	12.50	1.33	1.15	13.75	52.89	39.25	25.79
myciel5	6	Y	0.01	0.01	0.00	0.02	0.02	0.00	0.01	0.01	0.00	0.10	0.04	60.00
queen8_8	9	Y	1397.23	2144.76	−53.50	5.23	3.95	24.52	5.25	4.13	21.27	0.27	0.22	18.52
queen9_9	10	Y	–	–	–	8.22*	7.31*	10.95	8.43*	7.61*	9.68	56.91	45.51	20.03
queen11_11	13	Y	553.46	863.14	−55.95	15.13	9.71	35.79	14.47	11.63	19.59	205.66	175.62	14.61
queen14_14	17	Y	17.75	114.14	−543.04	8.27	11.82	−42.94	7.40	9.08	−22.75	943.30	708.92	24.86
queen8_12	12	Y	0.13	1.78	−1269.23	5.27	4.68	11.08	6.03	5.71	5.25	0.35	0.25	28.57

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

Table 3
Aggregate results (sum of solving times in seconds) for k-coloring (CPLEX)

Class	Instances		CPLEX			
	Solvable	Number	No delay	Delay	Saving	Saving %
Leighton	Y	3	4776.52	6660.07	−1883.55	−39.43%
(le graphs)	N	6	110.64	40.42	70.22	63.47%
	Total	9	4887.16	6700.49	−1813.33	−37.10%
Random	Y	1	19.22	18.15	1.07	5.57%
(DSJC graphs)	N	4	5026.08	4112.10	913.98	18.18%
	Total	5	5045.30	4130.25	915.05	18.14%
Reg. alloc.	Y	9	12423.85	11179.64	1244.21	10.01%
(fpsol, mulsol, zeroin graphs)	N	1	22.79	14.24	8.55	37.52%
	Total	10	12446.64	11193.88	1252.76	10.07%
SGB book	Y	4	2.16	2.27	−0.11	−5.09%
(anna, david, huck, jean graphs)	N	4	1.91	1.71	0.20	10.47%
	Total	8	4.07	3.98	0.09	2.21%
SGB miles	Y	3	138.24	197.23	−58.99	−42.67%
	N	2	2.98	2.07	0.91	30.54%
	Total	5	141.22	199.30	−58.08	−41.13%
SGB games	Y	1	0.97	0.88	0.09	9.28%
	N	1	0.97	1.05	−0.08	−8.25%
	Total	2	1.94	1.93	0.01	0.52%
SGB queen	Y	11	11806.53	6319.20	5487.33	46.48%
	N	4	8.93	9.33	−0.40	−4.48%
	Total	15	11815.46	6328.53	5486.93	46.44%
Mycielsky	Y	5	7.08	4.97	2.11	29.80%
	N	2	6.08	5.57	0.51	8.39%
	Total	7	13.16	10.54	2.62	19.91%
All	Y	37	29174.57	24382.41	4792.16	16.43%
	N	24	5180.38	4186.49	993.89	19.19%
	Total	61	34354.95	28568.90	5786.05	16.84%

ular, for each class, we write the number of instances solved, and the time needed by CPLEX for both positive and negative ones, with and without safe-delay (instances that couldn't be solved in one hour by both specifications have been removed). It can be observed that the reformulated specification is more efficient, on the average, especially on negative instances. Actually, safe-delay is really deleterious in only two cases: positive instances of "Leighton" and "SGB miles" classes.

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

SAT solving times are listed in Table 4 for different values for the deadline. Again, we have a mixed evidence for what concerns ZCHAFF, which benefits from safe-delay on many but not all instances, while savings in time are always positive when using local search solvers WALKSAT and BG-WALKSAT (even when they are not able to find a solution for positive instances, delaying constraints makes them terminate earlier). As for SATZ, savings in performances are often very high, even if this solver is able to solve only a small portion of the instance set. Interestingly, the blow-down in the number of clauses due to safe-delay, in some cases makes the solver able to handle some large instances (cf. Table 4, e.g., instance LA06 with deadline 50), preventing the system from running out of memory (even if, in many cases, the out-of-memory error changes to a timeout).

Table 4
Solving times (seconds) for job shop scheduling

Instance	Proc	Tasks	Jobs	Deadline	Solvable?	ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
						No delay	Delay	% sav.	No delay	Delay	% sav.	No delay	Delay	% sav.	No delay	Delay	% sav.
la01	5	50	10	33	N	0.69	0.85	−23.19	39.83	31.13	21.84	39.55	31.21	21.07	–	–	–
la01	5	50	10	34	Y	0.13	1.24	−853.85	40.77*	31.42*	22.94	40.72*	31.95*	21.53	154.49	1.76	98.86
la02	5	50	10	32	N	1.75	1.61	8.00	36.63	28.63	21.84	37.95	29.92	21.17	–	–	–
la02	5	50	10	33	Y	2.34	1.07	54.27	37.37*	28.80*	22.93	38.23*	29.10*	23.89	–	–	–
la03	5	50	10	31	N	2.77	2.10	24.19	32.78	25.37	22.62	32.92	25.11	23.70	–	–	–
la03	5	50	10	32	Y	1.92	0.31	83.85	32.85*	25.53*	22.27	32.67*	26.31*	19.44	–	–	–
la04	5	50	10	29	N	1.11	1.06	4.50	33.22	26.10	21.42	33.48	26.06	22.15	904.38	189.44	79.05
la04	5	50	10	30	Y	4.05	2.83	30.12	33.25*	26.22*	21.15	34.47*	26.02*	24.52	699.77	295.12	57.83
la05	5	50	10	28	N	1.33	0.99	25.56	28.92	22.83	21.04	29.25	23.10	21.03	–	–	–
la05	5	50	10	29	Y	0.29	0.57	−96.55	26.42	18.40	30.35	27.73*	19.46	29.81	0.89	0.58	34.83
la06	5	75	15	46	N	–	–	–	78.10	62.92	19.44	79.98	63.00	21.23	mem	–	–
la06	5	75	15	50	Y	0.44	1.89	−329.55	77.32	60.20*	22.14	78.77*	60.37*	23.36	mem	4.87	100.00
la07	5	75	15	43	N	–	–	–	78.35	62.21	20.59	80.10	62.55	21.91	–	–	–
la07	5	75	15	50	Y	1.27	0.98	22.83	78.62*	60.78*	22.68	78.08*	59.40*	23.93	mem	–	–
la08	5	75	15	42	N	–	–	–	77.02	60.72	21.16	76.18	60.17	21.02	–	–	–
la08	5	75	15	43	Y	262.44	181.53	30.83	76.83*	62.02*	19.28	77.25*	61.07*	20.95	–	–	–
la09	5	75	15	46	N	–	–	–	86.58	67.20	22.39	86.57	67.15	22.43	mem	–	–
la09	5	75	15	50	Y	0.31	5.76		85.73*	65.40*	23.72	83.65*	63.30	24.33	mem	3108.10	100.00
la10	5	75	15	46	N	–	–	–	80.80	64.86	19.72	81.42	63.07	22.54	mem	–	–
la10	5	75	15	50	Y	2.23	15.44		80.10*	61.90*	22.72	80.97*	61.05*	24.60	mem	–	–
la16	10	100	10	47	N	155.26	90.71	41.58	69.33	51.73	25.38	67.47	51.11	24.23	mem	–	–
la16	10	100	10	48	Y	151.64	18.71	87.66	69.87*	51.55*	26.22	70.70*	51.53*	27.11	mem	–	–
la17	10	100	10	39	N	5.42	3.72	31.37	57.03	44.46	22.03	56.23	43.78	22.14	–	–	–
la17	10	100	10	40	Y	3.68	4.19	−13.86	57.82*	44.20*	23.55	58.80*	43.71*	25.65	–	–	–
la18	10	100	10	41	N	6.31	4.21	33.28	62.30	47.53	23.70	61.12	45.48	25.58	–	–	–
la18	10	100	10	42	Y	5.75	3.27	43.13	63.72*	47.50*	25.45	62.17*	46.85*	24.64	–	–	–
la19	10	100	10	42	N	50.04	33.09	33.87	63.28	47.17	25.47	63.97	47.4	25.90	–	–	–
la19	10	100	10	43	Y	120.68	154.67	−28.17	64.83*	48.53*	25.14	63.52*	47.58*	25.09	–	3524.02	>2.11
la20	10	100	10	44	N	6.04	5.33	11.75	66.52	50.68	23.80	66.22	49.53	25.20	mem	–	–
la20	10	100	10	45	Y	20.86	13.91	33.32	68.87*	50.60*	26.52	67.47*	49.40*	26.78	mem	–	–
la22	10	150	15	48	N	–	–	–	mem	mem	–	mem	mem	–	mem	mem	–
la22	10	150	15	50	Y	1173.04	619.5	47.19	mem	mem	–	mem	mem	–	mem	mem	–
la23	10	150	15	50	N	–	934.79	>48.07	mem	mem	–	mem	mem	–	mem	mem	–
la23	10	150	15	53	Y	–	1028.56	42.86	mem	mem	–	mem	mem	–	mem	mem	–
la24	10	150	15	46	N	138.15	123.37	10.70	mem	mem	–	mem	mem	–	mem	mem	–
la24	10	150	15	48	Y	–	1083.31	>39.82	mem	mem	–	mem	mem	–	mem	mem	–
la25	10	150	15	48	N	356.56	344.02	3.52	mem	mem	–	mem	mem	–	mem	mem	–
la25	10	150	15	50	Y	438.27	510.21	−16.41	mem	mem	–	mem	mem	–	mem	mem	–
la29	10	200	20	50	N	705.09	216.74	69.26	mem	mem	–	mem	mem	–	mem	mem	–
la29	10	200	20	75	Y	–	1387.73	>22.90	mem	mem	–	mem	mem	–	mem	mem	–
la36	15	225	15	62	N	1238.04	782	36.84	mem	mem	–	mem	mem	–	mem	mem	–
la36	15	225	15	65	Y	–	527.86	>70.67	mem	mem	–	mem	mem	–	mem	mem	–
la38	15	225	15	58	N	–	844.58	>53.08	mem	mem	–	mem	mem	–	mem	mem	–
la38	15	225	15	75	Y	1329.74	–	<−35.36	mem	mem	–	mem	mem	–	mem	mem	–
la39	15	225	15	61	N	1193.76	1773.00	−48.52	mem	mem	–	mem	mem	–	mem	mem	–
la39	15	225	15	62	Y	787.54	918.24	−16.60	mem	mem	–	mem	mem	–	mem	mem	–
la40	15	225	15	59	N	989.98	712.96	27.98	mem	mem	–	mem	mem	–	mem	mem	–
la40	15	225	15	75	Y	1154.18	1256.54	−8.87	mem	mem	–	mem	mem	–	mem	mem	–

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

For what concerns the experiments in OPL instead, both CPLEX and SOLVER (run on the linear specification), seem not to be much affected by delaying constraints (or even affected negatively), and anyway are typically slower than SAT. For this reason, detailed results are omitted.

Summing up, we solved several thousands of instances. On the average, delaying constraints seems to be useful when using a SAT solver. In particular, local search solvers like WALKSAT and BG-WALKSAT almost always benefit

from the reformulation. The same happens when using SATZ. As for ZCHAFF, we have mixed evidence, since in some cases it seems not to be affected by safe-delay (cf., e.g., results in Table 1), or affected negatively, while in others it benefits from the reformulation (cf., e.g., Tables 2 and 4). The behavior of SAT solvers can be explained by considering the two phenomena that safe-delay produces: (i) The reduction of the number of clauses, and (ii) The enlargement of the set of solutions. The latter phenomenon is particularly beneficial when dealing with local search, as already observed in, e.g., [41,43] where symmetry-breaking (a technique that reduces the solution density) has been experimentally proven to be an obstacle for these algorithms. Of course, also the reduced number of clauses in general helps the solver. Nonetheless, it is worth noting that clauses derived from disjointness constraints are short, and this intuitively explains why the most efficient complete solver, ZCHAFF, that has powerful algorithms to efficiently deal with short clauses (with respect to SATZ), not always benefits from safe-delay.

As far as CPLEX and SOLVER are concerned, we have mixed evidence. First of all, as already observed, it is not always the case that the linear specification solved with CPLEX is always faster than the non-linear one solved with SOLVER, or vice versa. However, in those cases in which CPLEX is faster, delaying constraints often speeds-up the computation. This is consistent with the observation that, from a theoretical point of view, finding a partitioning is much harder than finding a covering, and in practice this often holds also in presence of additional constraints. Moreover, as for the beneficial behavior highlighted on negative instances, deeper analyses on the number of iterations and branches show that (i) More iterations are needed, on the average, by the simplex algorithm when invoked on the specification with no delay, and (ii) The number of branches is often unaffected when performing safe-delay. In particular, the latter issue suggests that each branch-and-bound subproblem is often solved more efficiently on the specification with safe-delay. Unfortunately, since CPLEX is not an open-source system, it is hard to analyze and explain its behavior in greater detail.

Finally, as for SOLVER when invoked on the linear specification, we observed that safe-delay has often (but not always) a negative effect, and this behavior is in line with the common observation, cf., e.g., the literature about symmetry-breaking and the addition of implied constraints, that *restricting* the set of solutions and adding tighter constraints helps when dealing with solvers based on backtracking, since this can significantly increase the amount of propagation, and consequently leads to a better pruning of the search space.

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 often results in faster solving. In this way, we have shown that reasoning on a specification can be very effective, at least on some classes of solvers.

Although Theorem 1 calls for a tautology checking (cf. Hyp 2), we have shown different specifications for which this test is immediate. Furthermore, we believe that, in practice, an automated theorem prover (ATP) can be used to reason on specifications, thus making it possible to automatically perform the task of choosing constraints to delay. Actually, in another paper [6] we have shown that state-of-the-art ATPs usually perform very well in similar tasks (i.e., detecting and breaking symmetries and detecting functional dependencies on problem specifications).

Related work. Several researchers addressed the issue of reformulation of a problem *after the instantiation phase*: as an example, in [50] it is shown how to translate an instantiated CSP into its Boolean form, which is useful for finding different reformulations, while in [13] the proposed approach is to generate a conjunctive decomposition of an instantiated CSP, by localizing independent subproblems. Furthermore, in [24], the system CGRASS is presented, allowing for the automated breaking of symmetries and the generation of useful implied constraints in a CSP. Finally, in [23] 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, i.e., regardless of the instance, 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., [25,49]. In particular, the idea of looking for "multivalued" functions has been already implemented in ad-hoc techniques for encoding various problems into SAT, like Graph coloring [42,46] and Job shop scheduling [15]. Conversely, in our work we take a specification-oriented approach, giving a formal justification of why some of the

constraints can be safely delayed, and presenting sufficient conditions that can be effectively used in order to isolate such constraints.

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., [2,10,30,44]). 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.

Since the effectiveness of a particular reformulation technique is expected to depend both on the problem and on the solver (even if this does not rule out, in principle, the possibility to find reformulations that are good for all solvers, or for solvers of a certain class), our research investigates the reformulation problem in different and complementary directions: in particular, in related work, we study how to detect symmetries [35] and functional dependencies [5] among predicates in specifications, and how specifications can be rewritten by exploiting these properties: symmetries can be broken by appropriate symmetry-breaking constraints added to the problem specification, and dependencies can be exploited by automatically synthesizing suitable search strategies. Experimental analysis shows how these approaches are effective in practice, for different classes of solvers. We have also shown [6] how automatic tools, such as first-order theorem provers and finite model finders can be exploited, in practical circumstances, to make this kind of reasoning by computer.

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., (9)), or to allow non-shrink second stages (cf., e.g., the specification for the Golomb ruler problem in Example 5), 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 is not always the case that delaying constraints, and so making the set of solutions larger, improves the solving process. Adding, e.g., symmetry-breaking 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 completely clear in which cases *removing* constraints results in better performances with respect to *adding* more constraints to the specification itself, even if it seems that an important role is played by the nature of constraints we remove or add, e.g., by their amenability to propagation in the search tree, together with the nature of the solver used, e.g., backtracking, linear, or based on local search. As for the latter class of solvers, it is known that enlarging the set of solutions can significantly speed-up performances (cf., e.g., [41,43]). Our experiments on WALKSAT and BG-WALKSAT confirm this thesis.

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

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". The authors are grateful to Carlo Mannino and Stefano Smriglio for useful discussions about CPLEX, and to the anonymous reviewers for their comments and suggestions.

Appendix A. Proofs of results

Proof of Theorem 1. Let $\mathcal{I}$ be an instance, M^* be a list of extensions for $(S_1, \ldots, S_h, S^*)$ such that $(M^*, \mathcal{I}) \models \psi^s$, and $ext(S)$ be an extension for S such that $(M^*, ext(S), \mathcal{I}) \models \psi^d$.

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

$$\big(M^*, ext(S), \mathcal{I}\big) \models \forall X\ S(X) \rightarrow S^*(X),$$

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

$$\left(M^*, ext(S), \mathcal{I}\right) \models \Xi. \tag{A.1}$$

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

$$\left(M^*, \mathcal{I}\right) \models \forall X\ \alpha(X) \rightarrow S^*(X),$$

and from Hyp 2 that:

$$(M^*, \mathcal{I}) \models \forall X\ \alpha(X) \rightarrow S^*(X) \wedge \beta(X).$$

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

$$\left(M^*, ext(S), \mathcal{I}\right) \models \forall X\ \alpha(X) \rightarrow S(X). \tag{A.2}$$

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

$$\left(M^*, ext(S), \mathcal{I}\right) \models \forall X\ S(X) \rightarrow \beta(X). \tag{A.3}$$

From (A.1–A.3), and from the observation that S^* does not occur in any of the right parts of them, the thesis follows. □

Proof of Theorem 2. Let $\mathcal{I}$ be an input instance, and M be a list of extensions for $(S_1, \ldots, S_h, S)$ such that $(M, \mathcal{I}) \models \psi$. Let S^* be defined in such a way that $ext(S^*) = ext(S)$.

Since solutions for ψ are also solutions for ψ^s, and since $ext(S^*) = ext(S)$, it follows that $((M - ext(S)) \cup ext(S^*))$ is a solution of ψ^s.

As for ψ^d, we observe that since M (which is a solution for the whole specification ψ) is also a solution for one of its constraints, namely $\forall X\ S(X) \rightarrow \beta(X)$ (the delayed constraint), then, from $ext(S^*) = ext(S)$, and in particular from the fact that $\forall X\ S(X) \rightarrow S^*(X)$ holds, it follows that $(M, ext(S^*)) \models \forall X\ S(X) \rightarrow S^*(X) \wedge \beta(X)$.

Conversely, from $ext(S^*) = ext(S)$, and in particular from the fact that $\forall X\ S^*(X) \rightarrow S(X)$ holds, it follows that $(M, ext(S^*)) \models \forall X\ S^*(X) \wedge \beta(X) \rightarrow S(X)$.

Hence, the thesis follows. □

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison Wesley Publishing Company, Reading, MA, 1995.

[2] A.V. Aho, Y. Sagiv, J.D. Ullman, Equivalences among relational expressions, SIAM Journal on Computing 2 (8) (1979) 218–246.

[3] E. Börger, E. Gräedel, Y. Gurevich, The Classical Decision Problem, Perspectives in Mathematical Logic, Springer, Berlin, 1997.

[4] C.A. Brown, L. Finkelstein, P.W. Purdom, Backtrack searching in the presence of symmetry, in: T. Mora (Ed.), Proceedings of the Sixth International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes, Rome, Italy, in: Lecture Notes in Computer Science, vol. 357, Springer, Berlin, 1988, pp. 99–110.

[5] M. Cadoli, T. Mancini, Exploiting functional dependencies in declarative problem specifications, in: Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA 2004), Lisbon, Portugal, in: Lecture Notes in Artificial Intelligence, vol. 3229, Springer, Berlin, 2004.

[6] M. Cadoli, T. Mancini, Using a theorem prover for reasoning on constraint problems, in: Proceedings of the Ninth Conference of the Italian Association for Artificial Intelligence (AI*IA 2005), Milano, Italy, in: Lecture Notes in Artificial Intelligence, vol. 3673, Springer, Berlin, 2005, pp. 38–49.

[7] M. Cadoli, T. Mancini, F. Patrizi, SAT as an effective solving technology for constraint problems, in: Proceedings of the Twentieth Convegno Italiano di Logica Computazionale (CILC 2005), Roma, Italy, 2005.

[8] M. Cadoli, A. Schaerf, Compiling problem specifications into SAT, Artificial Intelligence 162 (2005) 89–120.

[9] E. Castillo, A.J. Conejo, P. Pedregal, R. García, N. Alguacil, Building and Solving Mathematical Programming Models in Engineering and Science, John Wiley & Sons, 2001.

[10] A.K. Chandra, P.M. Merlin, Optimal implementation of conjunctive queries in relational databases, in: Proceedings of the Ninth ACM Symposium on Theory of Computing (STOC'77), Boulder, CO, ACM Press, 1977, pp. 77–90.

[11] P. Cheeseman, B. Kanefski, W.M. Taylor, Where the really hard problem are, in: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91), Sydney, Australia, Morgan Kaufmann, Los Altos, CA, 1991, pp. 163–169.

[12] B.M.W. Cheng, K.M.F. Choi, J.H.-M. Lee, J.C.K. Wu, Increasing constraint propagation by redundant modeling: an experience report, Constraints 4 (2) (1999) 167–192.

[13] B.Y. Choueiry, G. Noubir, On the computation of local interchangeability in discrete constraint satisfaction problems, in: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98), Madison, WI, AAAI Press/The MIT Press, 1998, pp. 326–333.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, 1990.
[15] J.M. Crawford, A.B. Baker, Experimental results on the application of satisfiability algorithms to scheduling problems, in: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94), Seattle, WA, AAAI Press/The MIT Press, 1994, pp. 1092–1097.
[16] J.M. Crawford, M.L. Ginsberg, E.M. Luks, A. Roy, Symmetry-breaking predicates for search problems, in: Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, MA, Morgan Kaufmann, Los Altos, CA, 1996, pp. 148–159.
[17] R. Dechter, Constraint networks (survey), in: Encyclopedia of Artificial Intelligence, second ed., John Wiley & Sons, 1992, pp. 276–285.
[18] D. East, M. Truszczyński, Predicate-calculus based logics for modeling and solving search problems ACM Transactions on Computational Logic, 2004, submitted for publication.
[19] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, in: R.M. Karp (Ed.), Complexity of Computation, American Mathematical Society, Providence, RI, 1974, pp. 43–74.
[20] P. Flener, Towards relational modelling of combinatorial optimisation problems, in: C. Bessière (Ed.), Proceedings of the International Workshop on Modelling and Solving Problems with Constraints, in conjunction with the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, 2001.
[21] P. Flener, J. Pearson, M. Ågren, Introducing ESRA, a relational language for modelling combinatorial problems, in: Proceedings of the Eighteenth IEEE Symposium on Logic in Computer Science (LICS 2004), Uppsala, Sweden, Springer, Berlin, 2003, pp. 214–232.
[22] R. Fourer, D.M. Gay, B.W. Kernigham, AMPL: A Modeling Language for Mathematical Programming, International Thomson Publishing, 1993.
[23] E.C. Freuder, D. Sabin, Interchangeability supports abstraction and reformulation for multi-dimensional constraint satisfaction, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97), Providence, RI, AAAI Press/The MIT Press, 1997, pp. 191–196.
[24] A. Frisch, I. Miguel, T. Walsh, CGRASS: A system for transforming constraint satisfaction problems, in: Proceedings of the Joint Workshop of the ERCIM Working Group on Constraints and the CologNet area on Constraint and Logic Programming on Constraint Solving and Constraint Logic Programming (ERCIM 2002), Cork, Ireland, in: Lecture Notes in Artificial Intelligence, vol. 2627, Springer, Berlin, 2002, pp. 15–30.
[25] A.M. Frisch, T.J. Peugniez, Solving non-boolean satisfiability problems with stochastic local search, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, Morgan Kaufmann, Los Altos, CA, 2001, pp. 282–290.
[26] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, San Francisco, CA, 1979.
[27] E. Giunchiglia, R. Sebastiani, Applying the Davis–Putnam procedure to non-clausal formulas, in: Proceedings of the Sixth Conference of the Italian Association for Artificial Intelligence (AI*IA'99), Bologna, Italy, in: Lecture Notes in Artificial Intelligence, vol. 1792, Springer, Berlin, 2000, pp. 84–94.
[28] F. Giunchiglia, T. Walsh, A theory of abstraction, Artificial Intelligence 57 (1992) 323–389.
[29] T. Hnich, T. Walsh, Why Channel? Multiple viewpoints for branching heuristics, in: Proceedings of the Second International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003), Kinsale, Ireland, 2003.
[30] A. Klug, On conjunctive queries containing inequalities, Journal of the ACM 1 (35) (1988) 146–160.
[31] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV System for knowledge representation and reasoning, in: ACM Transactions on Computational Logic, submitted for publication.
[32] C.M. Li, Integrating equivalency reasoning into Davis–Putnam procedure, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000), Austin, TX, AAAI Press/The MIT Press, 2000, pp. 291–296.
[33] C.M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Japan, Morgan Kaufmann, Los Altos, CA, 1997, pp. 366–371.
[34] T. Mancini, Reformulation techniques for a class of permutation problems, in: Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003), Kinsale, Ireland, in: Lecture Notes in Computer Science, vol. 2833, Springer, Berlin, 2003, p. 984.
[35] T. Mancini, M. Cadoli, Detecting and breaking symmetries by reasoning on problem specifications, in: Proceedings of the Sixth International Symposium on Abstraction, Reformulation and Approximation (SARA 2005), Airth Castle, Scotland, UK, in: Lecture Notes in Artificial Intelligence, vol. 3607, Springer, Berlin, 2005, pp. 165–181.
[36] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementation, John Wiley & Sons, 1990.
[37] B.D. McKay, Nauty user's guide (version 2.2). Available at http://cs.anu.edu.au/~bdm/nauty/nug.pdf, 2003.
[38] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, in: Proceedings of the Thirty Eighth Conference on Design Automation (DAC 2001), Las Vegas, NV, ACM Press, 2001, pp. 530–535.
[39] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (3,4) (1999) 241–273.
[40] C.H. Papadimitriou, Computational Complexity, Addison Wesley Publishing Company, Reading, MA, 1994.
[41] S.D. Prestwich, Supersymmetric modeling for local search, in: Proceedings of the Second International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002), Ithaca, NY, 2002.
[42] S.D. Prestwich, Local search on SAT-encoded colouring problems, in: Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Santa Margherita Ligure, Genova, Italy, in: Lecture Notes in Computer Science, vol. 2919, Springer, Berlin, 2004, pp. 105–119.

[43] S.D. Prestwich, A. Roli, Symmetry breaking and local search spaces, in: R. Barták, M. Milano (Eds.), Proceedings of the Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2005), Prague, CZ, in: Lecture Notes in Computer Science, vol. 3524, Springer, Berlin, 2005, pp. 273–287.
[44] Y. Sagiv, M. Yannakakis, Equivalence among relational expressions with the union and difference operations, Journal of the ACM 4 (27) (1980) 633–655.
[45] B. Selman, H.A. Kautz, B. Cohen, Local search strategies for satisfiability testing, in: M. Trick, D.S. Johnson (Eds.), Proceedings of the Second DIMACS Challenge on Cliques, Coloring, and Satisfiability, Providence, RI, 1993.
[46] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability instances, in: Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92), San Jose, CA, AAAI Press/The MIT Press, 1992.
[47] B.M. Smith, K. Stergiou, T. Walsh, Using auxiliary variables and implied constraints to model non-binary problems, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000), Austin, TX, AAAI Press/The MIT Press, 2000, pp. 182–187.
[48] P. Van Hentenryck, The OPL Optimization Programming Language, The MIT Press, 1999.
[49] T. Walsh, Permutation problems and channelling constraints, in: R. Nieuwenhuis, A. Voronkov (Eds.), Proceedings of the Eighth International Conference on Logic for Programming and Automated Reasoning (LPAR 2001), Havana, Cuba, in: Lecture Notes in Computer Science, vol. 2250, Springer, Berlin, 2001, pp. 377–391.
[50] R. Weigel, C. Bliek, On reformulation of constraint satisfaction problems, in: Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI'98), Brighton, UK, John Wiley & Sons, 1998, pp. 254–258.
[51] W. Zhang, A. Rangan, M. Looks, Backbone guided local search for maximum satisfiability, in: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, Morgan Kaufmann, Los Altos, CA, 2003, pp. 1179–1186.